María Alpuente
Germán Vidal (Eds.)

LNCS 5079

Static Analysis

15th International Symposium, SAS 2008
Valencia, Spain, July 2008
Proceedings

Springer

Lecture Notes in Computer Science 5079

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Alfred Kobsa
University of California, Irvine, CA, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
University of Dortmund, Germany
Madhu Sudan
Massachusetts Institute of Technology, MA, USA
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

María Alpuente   Germán Vidal (Eds.)

Static Analysis

15th International Symposium, SAS 2008
Valencia, Spain, July 16-18, 2008
Proceedings

Springer

Volume Editors

María Alpuente
Germán Vidal
Technical University of Valencia, DSIC
Camino de Vera S/N, 46022 Valencia, Spain
E-mail: {alpuente, gvidal}@dsic.upv.es

Library of Congress Control Number: 2008928275

CR Subject Classification (1998): D.3, F.3.1-2, I.2.2, F.4.2, B.8.1, D.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN	0302-9743
ISBN-10	3-540-69163-4 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-69163-1 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2008
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12277155 06/3180 5 4 3 2 1 0

Preface

Static analysis is a research area aimed at developing principles and tools for verification, certification, semantics-based manipulation, and high-performance implementation of programming languages and systems. The series of Static Analysis symposia has served as the primary venue for presentation and discussion of theoretical, practical, and application advances in the area.

This volume contains the papers accepted for presentation at the 15th International Static Analysis Symposium (SAS 2008), which was held July 16–18, 2008, in Valencia, Spain. The previous SAS conferences were held in Kongens Lyngby, Denmark (2007), Seoul, South Korea (2006), London, UK (2005), Verona, Italy (2004), San Diego, USA (2003), Madrid, Spain (2002), Paris, France (2001), Santa Barbara, USA (2000), Venice, Italy (1999), Pisa, Italy (1998), Paris, France (1997), Aachen, Germany (1996), Glasgow, UK (1995), and Namur, Belgium (1994).

In response to the call for papers, 63 contributions were submitted from 26 different countries. The Program Committee selected 22 papers, basing this choice on their scientific quality, originality, and relevance to the symposium. Each paper was reviewed by at least three Program Committee members or external referees. In addition to the contributed papers, this volume includes contributions by two outstanding invited speakers: Roberto Giacobazzi (Università degli Studi di Verona) and Ben Liblit (University of Wisconsin-Madison). The resulting volume offers the reader a complete landscape of the research in this area.

SAS 2008 was held concurrently with LOPSTR 2008, International Symposium on Logic–Based Program Synthesis and Transformation; PPDP 2008, ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming; and the SAS affiliated workshop PLID 2008, 4th International Workshop on Programming Language Interference and Dependence.

On behalf of the Program Committee, we would like to express our gratitude to all the authors who submitted papers and all external referees for their careful work in the reviewing process. The Program Chairs would like to thank in particular Alicia Villanueva (SAS Organizing Chair), Christophe Joubert (PPDP Organizing Chair), Josep Silva (LOPSTR Organizing Chair), and all the members of the Organization Committee who worked with enthusiasm in order to make this event possible. We are also grateful to Andrei Voronkov for making EasyChair available to us. Finally, we gratefully acknowledge the institutions that sponsored this event: Departamento de Sistemas Informáticos y Computación, EAPLS, ERCIM, Generalitat Valenciana, MEC (Feder) TIN2007-30509-E, and Universidad Politécnica de Valencia.

July 2008

María Alpuente
Germán Vidal

Organization

Program Chairs

María Alpuente	Technical University of Valencia, Spain
Germán Vidal	Technical University of Valencia, Spain

Program Committee

Elvira Albert	Complutense University of Madrid, Spain
Roberto Bagnara	University of Parma, Italy
Maurice Bruynooghe	Katholieke Universiteit Leuven, Belgium
Radhia Cousot	CNRS/École Polytechnique, France
Javier Esparza	Technical University of Munich, Germany
Sandro Etalle	University of Twente, The Netherlands
Moreno Falaschi	University of Siena, Italy
Stephen Fink	IBM T.J. Watson Research Center, New York, USA
John Gallagher	Roskilde University, Denmark
María del Mar Gallardo	University of Málaga, Spain
Chris Hankin	Imperial College, UK
Manuel Hermenegildo	Technical University of Madrid, Spain
Julia Lawall	University of Copenhagen, Denmark
Alexey Loginov	IBM T.J. Watson Research Center, New York, USA
Hanne Riis Nielson	Technical University of Denmark, Denmark
David Schmidt	Kansas State University, USA
Harald Sondergaard	University of Melbourne, Australia
Tachio Terauchi	Tohoku University, Japan
Ji Wang	National Lab. for Parallel and Distributed Processing, China

Steering Committee

Patrick Cousot	École Normale Supérieure, France
Gilberto Filé	Università di Padova, Italy
David Schmidt	Kansas State University, USA

Organizing Committee

Beatriz Alarcón, Gustavo Arroyo, Antonio Bella, Santiago Escobar, Vicent Estruch, Marco Feliu, César Ferri, Salvador Lucas, Raúl Gutiérrez, José Hernández, José Iborra, Christophe Joubert, Alexei Lescaylle, Marisa Llorens, Rafael Navarro, Pedro Ojeda, Javier Oliver, María José Ramírez, Daniel Romero, Josep Silva, Salvador Tamarit, Alicia Villanueva (Chair).

External Reviewers

Gianluca Amato
Puri Arenas
Demis Ballis
Maria Garcia de la Banda
Andrea Baruzzo
Joerg Bauer
Hubert Baumeister
Ralph Becket
Thomas Bolander
Rafael Caballero
Manuel Carro
Swarat Chaudhuri
Henning Christiansen
Robert Clariso
Michael Codish
Agostino Cortesi
Bart Demoen
Jérôme Feret
Maurizio Gabbrielli
Han Gao
Samir Genaim
Roberto Giacobazzi
Miguel Gomez-Zamalloa
Rene Rydhof Hansen
Jerry den Hartog
John Hatcliff
Fritz Henglein
Gerda Janssens
Bertrand Jeannet
Hugo Jonker
Stefan Kiefer
Herbert Kuchen
Vitaly Lagoon
Tal Lev-Ami
Pedro Lopez-Garcia
Michael Luttenberger
Damiano Macedonio
Angelika Mader
Julio Mariño
Matthieu Martel
Damien Masse'
Laurent Mauborgne
Guillaume Melquiond
Mario Mendez-Lojo
Maria Chiara Meo
Pedro Merino
Sebastian Nanz
Christoffer Rosenkilde Nielsen
Albert Nymeyer
Ricardo Peña
Quan Phan
David Pichardie
Henrik Pilegaard
Ernesto Pimentel
Christian Probst
Femke van Raamsdonk
Xavier Rival
Enric Rodriguez
Gwen Salaun
Sriram Sankaranarayanan
Peter Schachte
Tom Schrijvers
Stefan Schwoon
Helmut Seidl
Axel Simon
Stefano Soffia
Fred Spiessens
Fausto Spoto
Manu Sridharan
Peter Stuckey
Sriraman Tallam
Schrijvers Tom
Wim Vanhoof
Martin Vechev
Sven Verdoolaege
Vesal Vojdani
Pierre Wolper
Fan Yang
Hirotoshi Yasuoka
Ender Yuksel
Alessandro Zaccagnini
Enea Zaffanella
Damiano Zanardini

Table of Contents

Transforming Abstract Interpretations by Abstract Interpretation

New Challenges in Language-Based Security

Roberto Giacobazzi and Isabella Mastroeni

Dipartimento di Informatica - Università di Verona - Verona, Italy
{roberto.giacobazzi,isabella.mastroeni}@univr.it

Abstract. In this paper we exploit abstract interpretation for transforming abstract domains and semantics. The driving force in both transformations is making domains and semantics, i.e. abstract interpretations themselves, complete, namely precise, for some given observation. We prove that a common geometric pattern is shared by all these transformations, both at the domain and semantic level. This pattern is based on the notion residuated closures, which in our case can be viewed as an instance of abstract interpretation. We consider these operations in the context of language-based security, and show how domain and semantic transformations model security policies and attackers, opening new perspectives in the model of information flow in programming languages.

1 Introduction

Abstract interpretation [6] is not only a theory for the approximation of the semantics of dynamic systems, but also a way of thinking information and computation. From this point of view a program can be seen as an abstraction transformer, generalising Dijkstra's predicate transformer semantics, by considering abstractions as the objects of the computation: The way a program transforms abstractions tells us a lot about the way information flows and is manipulated during the computation. Abstract non-interference [13] is an example of this use of abstract interpretation, capturing precisely the intuition that in order to understand *who* can attack the code and *what* information flows, we have to consider programs as abstraction transformers, attackers as abstract interpretations, and secrets as data properties, which are abstractions again. This view lets out new possibilities for abstract interpretation use, e.g. in security, code design and obfuscation, as well as posing problems concerning the methods according to which these transformations are studied. Even if clearly previewed in the early stages of abstract interpretation [8], this approach to the use of abstract interpretation is still relatively unexplored.

In this paper we show that the standard theory of abstract interpretation, based on the so called adjoint-framework of Galois connections, can be directly applied to reason about operators that transform abstract domains and semantics, yet providing new formal methodologies for the systematic design of abstract

© Springer-Verlag Berlin Heidelberg 2008

domain refinements and program transformations (in our case program deformations). We first show that most domain transformers can be viewed as suitable problems of completeness w.r.t. some given semantic feature of the considered programming language. This observation has indeed an intuitive justification: The goal of refining a domain is always that of improving its precision with respect to some basic semantic operation (e.g., arithmetic operations, unification in logic programs, data structure operations in simple and higher-order types, and temporal operators in model checking). Analogously, simplifying domains corresponds to the dual operation of reducing precision with respect to analogous semantic operations. We show that most well known domain transformers can be interpreted in this way and that the relation between refinement and simplification on domains is indeed an instance of the same abstract interpretation framework lifted to higher types, i.e. where the objects of abstraction/concretization are abstract domains or semantics rather then computational objects.

$$\begin{tikzcd}
\text{Core} \arrow[rr, leftrightarrow, "+"{pos=0.05}, "-"{pos=0.95}] \arrow[d, leftrightarrow, "-"'{pos=0.05}, "+"'{pos=0.95}] & & \text{Expander} \\
\text{Shell} \arrow[rr, leftrightarrow, "-"'{pos=0.05}, "+"'{pos=0.95}] & & \text{Compressor}
\end{tikzcd}$$

Residuated closures [2,22] provide the instance of Galois connections on closure operators, i.e. on abstract domains. We prove that standard refinement/simplification (also called shell/core) and expansion/compression are residuated closures on abstract domains. In particular it is always possible to derive an operation which reduces a given domain by considering either the right (+) or left (−) adjoint of a domain refinement. We show that the meaning of these transformations are deeply different: while the left adjoint of a refinement is always a simplification which keeps completeness (core), the right one moves towards maximal incompleteness by reducing the domain (compression). Similarly we can always refine a domain by considering the same adjoints of a domain simplification. In this case the left adjoint always moves towards maximal incompleteness by refining the domain (expander) while the right one refines the domain yet keeping completeness (shell), as depicted above. We prove that this construction can be generalised to arbitrary semantic transformers, which in view of [10] may correspond to code transformations, where instead of transforming domains we transform semantics. The result is a unique geometric interpretation of abstract domain and semantic transformers where both notions can be systematically derived by considering standard abstract interpretation methods. We apply these transformations to the case of language-based security, by modelling security policies and attackers as suitable abstract domain and semantic transformations.

2 Abstract Domains Individually and Collectively

We consider standard abstract domain definition as formalised in [6] and [8] in terms of Galois connections. It is well known that this is a restriction for abstract interpretation because relevant abstractions do not form Galois connections and Galois connections are not expressive enough for modelling dynamic fix-point approximation [9]. Formally, if $\langle C, \leq, \top, \bot, \vee, \wedge \rangle$ is a complete lattice, monotone functions $\alpha : C \xrightarrow{\text{m}} A$ and $\gamma : A \xrightarrow{\text{m}} C$ form an *adjunction* or a *Galois connection* if for any $x \in C$ and $y \in A$: $\alpha(x) \leq_A y \Leftrightarrow x \leq_C \gamma(y)$. α [resp. γ] is the *left-* [*right-*]*adjoint* to γ [α] and it is additive [co-additive], i.e. it preserves *lub*'s [*glb*] of all subsets of the domain (emptyset included). The right adjoint of a function α is $\alpha^+ \stackrel{\text{def}}{=} \lambda x.\ \bigvee \{\ y \mid \alpha(y) \leq x\ \}$. Conversely the left adjoint of γ is $\gamma^- \stackrel{\text{def}}{=} \lambda x.\ \bigwedge \{\ y \mid x \leq \gamma(y)\ \}$ [8]. Abstract domains can be also equivalently formalized as closure operators on the concrete domain. An *upper [lower] closure operator* $\rho : P \longrightarrow P$ on a poset P is monotone, idempotent, and extensive: $\forall x \in P.\ x \leq_P \rho(x)$ [reductive: $\forall x \in P.\ x \geq_P \rho(x)$]. Closures are uniquely determined by their fix-points $\rho(C)$. The set of all upper [lower] closure operators on P is denoted by $uco(P)$ [$lco(P)$]. The *lattice of abstract domains* of C, is therefore isomorphic to $uco(C)$, (cf. [6, Section 7] and [8, Section 8]). Recall that if C is a complete lattice, then $\langle uco(C), \sqsubseteq, \sqcup, \sqcap, \lambda x.\top, id\rangle$ is a complete lattice [24], where $id \stackrel{\text{def}}{=} \lambda x.x$ and for every $\rho, \eta \in uco(C)$, $\rho \sqsubseteq \eta$ iff $\forall y \in C.\ \rho(y) \leq \eta(y)$ iff $\eta(C) \subseteq \rho(C)$. A_1 is more precise than A_2 (i.e., A_2 is an abstraction of A_1) iff $A_1 \sqsubseteq A_2$ in $uco(C)$. Given $X \subseteq C$, the least abstract domain containing X is the least closure including X as fix-points, which is the *Moore-closure* $\mathcal{M}(X) \stackrel{\text{def}}{=} \{\bigwedge S \mid S \subseteq X\}$. Precision of an abstract interpretation typically relies upon the structure of the abstract domain [19]. Depending on where we compare the concrete and the abstract computations we obtain two different notions of completeness. If we compare the results in the abstract domain, we obtain what is called *backward completeness* ($\mathscr{B}$), while, if we compare the results in the concrete domain we obtain the so called *forward completeness* ($\mathscr{F}$) [8,19,15]. Formally, if $f : C \xrightarrow{\text{m}} C$ and $\rho \in uco(C)$, then ρ is $\mathscr{B}$-complete if $\rho \circ f \circ \rho = \rho \circ f$, while it is $\mathscr{F}$-complete if $\rho \circ f \circ \rho = f \circ \rho$. The problem of making abstract domains $\mathscr{B}$-complete has been solved in [19] and later generalised to $\mathscr{F}$-completeness in [15]. In a more general setting let $f : C_1 \longrightarrow C_2$ and $\rho \in uco(C_2)$ and $\eta \in uco(C_1)$. $\langle \rho, \eta\rangle$ is a pair of $\mathscr{B}$[$\mathscr{F}$]-complete abstractions for f if $\rho \circ f = \rho \circ f \circ \eta$ [$f \circ \eta = \rho \circ f \circ \eta$]. A pair of domain transformers has been associated with any completeness problem [11,16], which are respectively a *domain refinement* and *simplification*. In [19] and [15], a constructive characterization of the most abstract refinement, called *complete shell*, and of the most concrete simplification, called *complete core*, of any domain, making it $\mathscr{F}$ or $\mathscr{B}$-complete, for a given continuous function f, is given as a solution of simple domain equations given by the following basic operators:

$$\boxed{\begin{array}{r|l} R_f^{\mathscr{F}} \stackrel{\text{def}}{=} \lambda X.\, \mathcal{M}(f(X)) & R_f^{\mathscr{B}} \stackrel{\text{def}}{=} \lambda X.\, \mathcal{M}(\bigcup_{y \in X} \max(f^{-1}(\downarrow y))) \\ C_f^{\mathscr{F}} \stackrel{\text{def}}{=} \lambda X.\, \{\ y \in L \mid f(y) \subseteq X\ \} & C_f^{\mathscr{B}} \stackrel{\text{def}}{=} \lambda X.\, \{\ y \in L \mid \max(f^{-1}(\downarrow y)) \subseteq X\ \} \end{array}}$$

Let $\ell \in \{\mathscr{F}, \mathscr{B}\}$. In [19] the authors proved that the most concrete $\beta \sqsupseteq \rho$ such that $\langle \beta, \eta\rangle$ is ℓ-complete and the most abstract $\beta \sqsubseteq \eta$ such that $\langle \rho, \beta\rangle$ is ℓ-complete are respectively the ℓ-complete core and ℓ-complete shell, which are: $\mathcal{C}_f^{\ell,\eta}(\rho) \stackrel{\text{def}}{=} \rho \sqcup C_f^{\ell}(\eta)$ and $\mathcal{R}_f^{\ell,\rho}(\eta) \stackrel{\text{def}}{=} \eta \sqcap R_f^{\ell}(\rho)$. When $\eta = \rho$, then the fix-point iteration on abstract domains of the above function $\mathcal{R}_f^{\ell}(\rho) = gfp(\lambda X.\ \rho \sqcap R_f^{\ell}(X))$ is called the *absolute ℓ-complete shell*. By construction if f is additive then

$\mathcal{R}_f^{\mathscr{B}} = \mathcal{R}_{f^+}^{\mathscr{F}}$ (analogously $\mathcal{C}_f^{\mathscr{B}} = \mathcal{C}_{f^+}^{\mathscr{F}}$) [15]. This means that when we have to solve a problem of $\mathscr{B}$-completeness for an additive function then we can equivalently solve the corresponding $\mathscr{F}$-completeness problem for the right adjoint function.

Completeness can be used for modelling non-interference in language-based security [14,1]. Abstract non-interference (ANI) [13] is a natural weakening of non-interference by abstract interpretation. Let $\eta, \rho \in uco(\mathbb{V}^{\mathtt{L}})$ and $\phi \in uco(\mathbb{V}^{\mathtt{H}})$, where $\mathbb{V}^{\mathtt{L}}$ and $\mathbb{V}^{\mathtt{H}}$ are the domains of public ($\mathtt{L}$) and private ($\mathtt{H}$) variables. η and ρ characterise the *attacker* to the policy. A policy characterises a weakening of the information that can flow. We consider $\phi \in uco(\mathbb{V}^{\mathtt{H}})$, which states what, of the private data, can indeed flow to the output observation, the so called *declassification* of ϕ. In the following if P is a program $[\![P]\!]$ denotes its denotational semantics. A program P satisfies declassified ANI if $\forall h_1, h_2 \in \mathbb{V}^{\mathtt{H}}, \forall l_1, l_2 \in \mathbb{V}^{\mathtt{L}}$:

$$\eta(l_1) = \eta(l_2) \ \wedge \ \phi(h_1) = \phi(h_2) \ \Rightarrow \ \rho([\![P]\!](h_1, \eta(l_1))^{\mathtt{L}}) = \rho([\![P]\!](h_2, \eta(l_2))^{\mathtt{L}}).$$

This notion says that, whenever the attacker is able to analyze the input property η and the ρ property of the output, then it can observe nothing more than the property ϕ of private input. Clearly, transforming abstractions corresponds here to transform attackers and policies. If $\mathcal{H}_\rho(\langle x^{\mathtt{H}}, x^{\mathtt{L}}\rangle) \overset{\mathrm{def}}{=} \langle \mathbb{V}^{\mathtt{H}}, \rho(x^{\mathtt{L}})\rangle$, $\mathcal{H}_\eta^\phi(\langle x^{\mathtt{H}}, x^{\mathtt{L}}\rangle) \overset{\mathrm{def}}{=} \langle \phi(x^{\mathtt{H}}), \eta(x^{\mathtt{L}})\rangle$, $[\![P]\!]_\eta \overset{\mathrm{def}}{=} \lambda x.\, [\![P]\!](x^{\mathtt{H}}, \eta(x^{\mathtt{L}}))$ and the weakest liberal precondition semantics is $wlp_\eta^P \overset{\mathrm{def}}{=} \lambda X.\ \{\ \langle h, \eta(l)\rangle \mid [\![P]\!](\langle h, \eta(l)\rangle) \subseteq X\ \}$ [5], then the equivalent $\mathscr{B}$- and $\mathscr{F}$-completeness equations modelling ANI above, with attacker η and ρ, and declassified by the partitioning abstraction[1] ϕ are [14]:

$$\mathcal{H}_\rho \circ [\![P]\!]_\eta \circ \mathcal{H}_\eta^\phi = \mathcal{H}_\rho \circ [\![P]\!] \iff \mathcal{H}_\eta^\phi \circ wlp_\eta^P \circ \mathcal{H}_\rho = wlp_\eta^P \circ \mathcal{H}_\rho \qquad (1)$$

3 The Geometry of Abstract Domain Transformers

The notion of abstract domain refinement and simplification has been introduced in [11,16] as a generalisation of most well-known operations for transforming abstract domains, e.g., those introduced in [8]. In this section we consider these notions as instances of a more general pattern where abstract domain transformers have the same structures of abstract domains. For the sake of simplicity we consider unary functions only, even if all the following results can be easily generalized to generic n-ary functions (e.g., see [16] for examples). If $\tau, \eta : uco(C) \longrightarrow uco(C)$, following [16], we distinguish between *domain refinements* which concretise domains, i.e. $X \subseteq \tau(X)$, and *domain simplifications* which simplify domains, i.e. $\eta(X) \subseteq X$. Monotone refinements and simplifications can be associated with closure operators: If τ $[\eta]$ is a monotone refinement [simplification] then $\lambda X.\, gfp(\lambda Y.\, X \sqcap \tau(Y))$ $[\lambda X.\, lfp(\lambda Y.\, X \sqcup \eta(Y))]$ is the corresponding idempotent refinement [simplification] [7]. Therefore, monotone refinements τ and simplifications η may have the same structure of abstract domains, as closure operators on $uco(C)$, resp. $\tau \in lco(uco(C))$ and

1 An abstraction of a complete Boolean concrete domain C is *partitioning* if it is a complete Boolean sub-semilattice of C.

$\eta \in uco(uco(C))$. This observation will be the basis in order to lift standard abstract interpretation in higher types, i.e. from a theory for approximating computational objects, such as semantics, to a theory for abstract domain transformers. In this sense, standard Cousot and Cousot's Galois connection based abstract interpretation theory is perfectly adequate to develop also a theory of abstract domain transformers providing these transformations with the same calculational design techniques which are known for standard abstract interpretation [4]. In particular, Janowitz [2,22], characterised the structure of residuated (adjoint) closure operators by the following basic result.

Theorem 1 ([22]). *Let $\langle \eta, \eta^+ \rangle$ and $\langle \eta^-, \eta \rangle$ be pairs of adjoint operators on C.*

$$(1) \quad \eta \in uco(C) \;\Leftrightarrow\; \eta^+ \in lco(C) \;\Leftrightarrow\; \begin{cases} \eta \circ \eta^+ = \eta^+ \\ \eta^+ \circ \eta = \eta \end{cases}$$

$$(2) \quad \eta \in uco(C) \;\Leftrightarrow\; \eta^- \in lco(C) \;\Leftrightarrow\; \begin{cases} \eta \circ \eta^- = \eta \\ \eta^- \circ \eta = \eta^- \end{cases}$$

Stated in terms of refinements, this result says that any (either right or left) adjoint of a refinement [simplification] is a simplification [refinement]. This means that for any refinement [simplification] we may have two possible simplifications [refinements] corresponding to either right or left adjoint, when they exist. Let $\tau \in lco(C)$ be a domain refinement. By Th. 1, if τ^- exists then $\tau^-(\tau(X)) = \tau(X)$ and $\tau(\tau^-(X)) = \tau^-(X)$. This means that τ^- is a simplification such that both τ and τ^- have the same sets of fix-points, namely τ^- reduces any abstract domain X until the reduced domain Y satisfies $\tau(Y) = Y$. Due to the analogy with completeness, in this case we call τ^- the *core* of τ and τ the *shell* of τ^-.

Proposition 2. *Let $\tau \in lco(C)$ and $\eta \in uco(C)$. If $\langle \tau^-, \tau \rangle$ and $\langle \eta, \eta^+ \rangle$ are pairs of adjoint functions then we have $\tau^- = \lambda X.\ \bigwedge\{\tau(Y) | \tau(Y) \leq X\}$ and $\eta^+ = \lambda X.\ \bigvee\{\eta(Y) | X \leq \eta(Y)\}$.*

The interpretation of the second point of Th. 1 for refinements, i.e, of the right adjoint of a refinement τ, when it exists, is quite different. By Th. 1, we have that if τ^+ exists then $\tau^+(\tau(X)) = \tau^+(X)$ and $\tau(\tau^+(X)) = \tau(X)$. In this case $\tau^+(X)$ is not a fix-point of τ. Instead, it returns the most abstract domain whose precision can be lifted to that of X by refinement. τ^+ reduces any abstract domain X such that $\tau(X) = X$ towards the most abstract domain Y such that $\tau(Y) = X$. We call τ^+ the *compressor* of τ and τ the *expander* of τ^+.

Proposition 3. *Let $\tau \in lco(C)$ and $\eta \in uco(C)$. If $\langle \tau, \tau^+ \rangle$ and $\langle \eta^-, \eta \rangle$ are pairs of adjoint functions then we have $\tau^+ = \lambda X.\ \bigvee\{Y | \tau(Y) = \tau(X)\}$ and $\eta^- = \lambda X.\ \bigwedge\{Y | \eta(X) = \eta(Y)\}$.*

3.1 Shell vs. Core

Not all domain transformers admit adjoints, because not all closure are either additive or co-additive functions. However adjointness can be weakened by considering only those properties that make a transformer reversible, either as a pair shell-core or expander-compressor. In the following we describe the properties of invertible refinements since the properties of invertible simplifications can be derived by duality as shown above. By Prop. 2, the relation between the shell τ and the core τ^- is characterized by the fact that $\tau^-(X)$ isolates the most concrete domain which is contained in X and which is a fix-point of τ: $\tau^-(X) = \bigwedge\{\, \tau(Y) \mid X \leq \tau(Y) \,\}$. While $\tau^- \circ \tau = \tau$ always holds for any $\tau \in lco(C)$, the key property which characterizes the pair shell-core, $\tau \circ \tau^- = \tau^-$, holds iff $\langle \tau^-, \tau \rangle$ is a pair of adjoint functions.

Theorem 4. *Let $\tau \in lco(C)$. $\tau \circ \tau^- = \tau^-$ holds iff τ is co-additive.*

This means that the relation between shell and core just holds in the standard adjoint framework. An example of the core/shell is in $\mathscr{F}$-completeness, where $\mathcal{C}_f^{\mathscr{F}}$ is a $\mathscr{F}$-completeness core for f iff $\mathcal{C}_f^{\mathscr{F}}(X) \sqsubseteq Y \Leftrightarrow X \sqsubseteq \mathcal{R}_f^{\mathscr{F}}(Y)$ [15].

3.2 Expander vs. Compressor

The notion of domain compression was introduced in [11,16] and later developed for the case of *disjunctive completion* $\lambda X. \curlyvee(X)$, making a domain a complete join-subsemilattice of the concrete domain (viz. an additive closure), and *reduced product*, which is the *glb* in *uco*(C), resp. in [17] and [3]. In view of Janowitz's results we review the theory of domain compression in [18], where the notion of *uniform closure* was introduced. $f : C \longrightarrow C$ is *join-uniform* [18] if for all $Y \subseteq C$, $(\exists \bar{x} \in Y.\, \forall y \in Y.\, f(y) = f(\bar{x})) \Rightarrow (\exists \bar{x} \in Y.\, f(\bigvee Y) = f(\bar{x}))$. Meet-uniformity is defined dually. By Prop. 3, the relation between the expander τ and its compressor τ^+ is characterized by the fact that $\tau^+(X)$ is the most abstract domain which allows us to reconstruct $\tau(X)$ by refinement: $\tau^+(X) = \bigsqcup\{\, Y \mid \tau(X) = \tau(Y) \,\}$. While $\tau^+ \circ \tau = \tau^+$ always holds for any $\tau \in lco(C)$, the key property which characterizes the pair expander/compressor, is $\tau \circ \tau^+ = \tau$ and it holds iff τ is join-uniform. Join-uniformity captures precisely the intuitive insight of the pair expander/compressor. If τ is join-uniform, and $x \in C$, then there always exists a (unique) element $\bigvee Z$, such that $\tau(\bigvee Z) = \tau(x)$ where $Z = \{y \in C \mid \tau(x) = \tau(y)\}$. As observed in [18], τ^+ may fail monotonicity. In [18] the authors proved that τ^+ is monotone on a lifted order induced by τ. Let $\tau : C \xrightarrow{\text{m}} C$, the lifted order $\leq_\tau \subseteq C \times C$ is defined as follows: $x \leq_\tau y \;\Leftrightarrow\; (\tau(x) \leq \tau(y)) \;\wedge\; (\tau(x) = \tau(y) \;\Rightarrow\; x \leq y)$. $\leq_\tau$ is such that $\leq \Rightarrow \leq_\tau$. Next theorem strengthen [18, Th. 5.10][2] proving the equivalence between reversibility and adjointness in $\leq_\tau$ for any refinement.

Theorem 5. *Let $\tau \in lco(L)$ and $\tau^+ = \lambda x. \bigvee\{\, y \mid \tau(y) = \tau(x) \,\}$. The following facts are equivalent:*

1. *$\tau \circ \tau^+ = \tau$;*
2. *τ is join-uniform on $\leq$;*
3. *τ is additive on $\leq_\tau$ and the right adjoint of τ on $\leq_\tau$ is τ^+.*

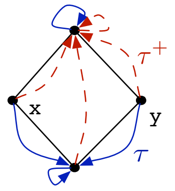
[2] In [18, Th. 5.10] the authors proved only that 1. $\Leftrightarrow$ 2. $\Rightarrow$ 3.

The relation between join-uniformity and meet-uniformity is preserved by the relation of adjointness.

Proposition 6. *Let $\tau \in lco(L)$ be a join-uniform operator on $\leq$. Then we have $\tau^+(x) = \bigvee\{y | \tau(x) = \tau(y)\}$ is meet-uniform on $\leq$.*

Unfortunately, the inverse implication of Prop. 6 does not hold in general.

Example 7. In the picture on the right, we provide an example where the map τ^+ (denoted with dashed arrows) is meet-uniform, while τ is not join-uniform. Indeed, note that $\tau^+ = \lambda X. \top$, which is clearly meet-uniform, while τ is not join-uniform since, for instance, $\tau(x) = \tau(y) \neq \top$, but $\tau(x \vee y) = \tau(\top) = \top$.



Examples of join-uniform refinements include disjunctive completion and reduced product , where the corresponding compressors are the *disjunctive base* [17] and *complementation* [3]. Both are compressors associated with completeness refinements, the first being $\mathscr{F}$-completeness w.r.t. disjunction and the second being $\mathscr{B}$-completeness w.r.t. conjunction, i.e. $\mathscr{F}$-completeness w.r.t. implication [21,20]. The problem of studying whether a generic completeness refinement admits a compressor has been investigated with the aim of finding a characterization of all the functions f such that the corresponding completeness refinement has the compressor. The only known characterisation is in [12], for the case of $\mathscr{F}$-completeness. In this case the authors provide an algorithmic construction which is based on the the notion of f-reducible element, i.e. those elements that can be generated by others by means of the function f or by Moore closure. If all the f-irreducible elements form an abstract domain, then this is called the *complete base* and the compressor locally (i.e., for the particular abstract domain to which we apply the algorithm) exists.

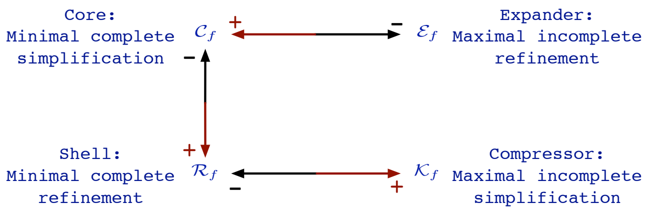


Fig. 1. Basic abstract domain transformers

3.3 Transforming Abstractions for Transforming Policies

By transforming abstractions in abstract non interference (Eq. 1), we transform the corresponding non-interference policy. In particular, if we transform the input abstraction we transform the declassification policy, while when we transform the output abstraction we transform the attacker policy [14]. Let us see the meaning of the shell/core and expander/compressor transformations in these cases.

Shell: The Maximal Released Information by an Attacker. The shell minimally refines the domain $\mathcal{H}_\eta^\phi$ in order to satisfy Eq. 1 [14]. When the equation does not hold, it means that the given attacker is able to disclose more information than what is modelled by ϕ about the private input. In non-interference, disclosing means observing variations in the ρ abstraction of the output due to input variations not modelled by ϕ. In other words, there are at least two private inputs h_1 and h_2 with the same property ϕ which generate a different ρ property in output. Therefore in order to characterise the closest declassification policy satisfied by the program, namely the maximal information disclosed by the attacker, we have to refine the policy ϕ, and in particular we have to distinguish by ϕ the values h_1 and h_2, above. This is what the shell does, by modelling the minimal amount of distinguishable values that the attacker is able to observe. We denote by $\mathcal{R}_{\llbracket P \rrbracket}^{\mathcal{H}_\rho}(\mathcal{H}_\eta^\phi)$ this transformation.

Example 8. Consider the program fragment: $P \stackrel{\text{def}}{=} l := l * h^2$, with $l : \mathtt{L}$ and $h : \mathtt{H}$. We want to find the shell in order to make the input/outpur pair of abstract domains $\langle \mathcal{H}, \mathcal{H}_{\text{Par}} \rangle$, where $Par \stackrel{\text{def}}{=} \{\mathbb{Z}, 2\mathbb{Z}+1, 2\mathbb{Z}, \varnothing\}$, complete for $\llbracket P \rrbracket$. Let $\mathcal{H} \stackrel{\text{def}}{=} \mathcal{H}_{id}^{id}$.

$$\mathcal{R}_{\llbracket P \rrbracket}^{\mathcal{H}_{\text{Par}}}(\mathcal{H}) = \mathcal{H} \sqcap \left(\left\{ \begin{array}{l} \langle \mathbb{Z}, \mathbb{Z} \rangle, \langle \mathbb{Z}, 2\mathbb{Z} \rangle \cup \langle 2\mathbb{Z}, 2\mathbb{Z}+1 \rangle, \langle 2\mathbb{Z}+1, 2\mathbb{Z}+1 \rangle, \\ \langle 2\mathbb{Z}+1, 2\mathbb{Z} \rangle, \varnothing \end{array} \right\} \right)$$

Note that $\langle 2\mathbb{Z}, 2\mathbb{Z}+1 \rangle, \langle 2\mathbb{Z}, l \rangle \in \mathcal{R}_{\llbracket P \rrbracket}^{\mathcal{H}_{\text{Par}}}(\mathcal{H})$ for each $l \in 2\mathbb{Z}+1$. For instance, in this case $\mathcal{H}_{\text{Par}}(\llbracket P \rrbracket(\mathcal{R}_{\llbracket P \rrbracket}^{\mathcal{H}_{\text{Par}}}(\mathcal{H})(\langle 2, 3 \rangle))) = \mathcal{H}_{\text{Par}}(\llbracket P \rrbracket(\langle 2\mathbb{Z}, 3 \rangle)) = \langle \mathbb{Z}, 2\mathbb{Z} \rangle = \mathcal{H}_{\text{Par}}(\llbracket P \rrbracket(\langle 2, 3 \rangle))$.

Core: The Most Powerful Attacker for a Declassification Policy. The core minimally transform the domain $\mathcal{H}_\rho$ in order to satisfy Eq. 1 [14]. Exactly as before, when the equation does not hold, it means that the attacker, observing ρ, is able to disclose more information than what is modelled by ϕ about the private input. In this case we simplify the model of the attacker. If there are at least two private inputs h_1 and h_2 having the same property ϕ which generate a different ρ property in output, then we decide to simplify the attacker making the corresponding output values indistinguishable. The core of $\mathcal{H}_\rho$ collapses all the outputs generated by private inputs with a different ϕ property. In this way, we are able to characterize, given a declassification policy ϕ, the most powerful attacker, weaker than ρ, which is harmless, namely unable to disclose any information about ϕ of private data. We denote by $\mathcal{C}_{\llbracket P \rrbracket}^{\mathcal{H}_\eta^\phi}(\mathcal{H}_\rho)$ this tranformation.

Example 9. Let the program $P \stackrel{\text{def}}{=}$ **while** $h \neq 0$ **do** $l := 2l; h := 0$ **endw**, with $l : \mathtt{L}$ and $h : \mathtt{H}$. $\mathcal{C}_{\llbracket P \rrbracket}^{\mathcal{H}}(\mathcal{H}_{id}) = \{\langle \mathbb{Z}, L \rangle \mid \forall l \in \mathbb{V}^{\mathtt{L}} \,.\, l \in L \Leftrightarrow 2l \in L\}$ makes $\langle \mathcal{H}, \mathcal{H}_{id} \rangle$ complete for $\llbracket P \rrbracket$. It is easy to show that $\mathcal{C}_{\llbracket P \rrbracket}^{\mathcal{H}}(\mathcal{H}_{id})$ is the abstract domain $\Upsilon\left(\{n\{2\}^{\mathbb{N}} \mid n \in 2\mathbb{Z}+1\}\right)$, where $\{2\}^{\mathbb{N}} \stackrel{\text{def}}{=} \{2^k \mid k \in \mathbb{N}\}$. Then $\mathcal{C}_{\llbracket P \rrbracket}^{\mathcal{H}}(\mathcal{H}_{id})(\llbracket P \rrbracket(\mathcal{H}(\langle 3, 5 \rangle))) = \mathcal{C}_{\llbracket P \rrbracket}^{\mathcal{H}}(\mathcal{H}_{id})(\llbracket P \rrbracket(\mathbb{Z}, 5)) = \mathcal{C}_{\llbracket P \rrbracket}^{\mathcal{H}}(\mathcal{H}_{id})(\langle \mathbb{Z}, \{5, 10\} \rangle) = \langle \mathbb{Z}, 5\{2\}^{\mathbb{N}} \rangle$, $\mathcal{C}_{\llbracket P \rrbracket}^{\mathcal{H}}(\mathcal{H}_{id})(\llbracket P \rrbracket(\langle 3, 5 \rangle)) = \mathcal{C}_{\llbracket P \rrbracket}^{\mathcal{H}}(\mathcal{H}_{id})(\langle \mathbb{Z}, \{10\} \rangle) = \langle \mathbb{Z}, 5\{2\}^{\mathbb{N}} \rangle$.

Expander. Let us consider the left adjoint of the core. In this case we look for the more concrete attacker, which adjoints the same harmless one. This is interpreted by saying that the expander provides the inferior limit to the range of all the attacks making a given policy insecure. In other words it provides the most successful attack for the given policy.

Example 10. Consider the program fragment $P \stackrel{\text{def}}{=} l := 2h$. We can easily show that the most powerful harmless attacker is the one that cannot distinguish even numbers, i.e., $\Upsilon(\{2\mathbb{Z}, \{1\}, \{3\}, \ldots\})$. Suppose now that the inital observer of the program in output observes $\rho = \{\mathbb{Z}, 2\mathbb{Z} \smallsetminus \{0\}, \{0\}, 2\mathbb{Z}+1, \varnothing\}$. Then we obtain that the most powerful harmless attacker which is more abstract than ρ is *Par*. At this point the compressor provides the most concrete abstraction, whose core is exactly *Par*. We can show that this abstraction is $\Upsilon(\{2\mathbb{Z}+1, \{0\}, \{2\}, \{4\}, \ldots, \})$. We can interpret this abstraction as the most powerful malicious attacker, namely the one that is able to exploit as much as possible the failure of the non-interference policy, since it can disclose the exact value of h. Any more abstract domain, has to confuse some even numbers, for instance it can confuse 0 with 2, which means that it cannot distinguish when $h = 0$ and $h = 1$.

Compressor. Finally, let us consider the right adjoint of the completeness shell. In this case we look for the most abstract declassification policy which cannot capture what is indeed released by the attacker observing the program. Also in this case, we interpret this abstraction as a superior limit to the range of all the policies which are inadequate to protect a program from an attacker.

Example 11. Consider the program $P \stackrel{\text{def}}{=} \textbf{if } h = 0 \textbf{ then } l := 0 \textbf{ else } l := |l|(h/|h|)$, where $|x|$ is the absolute value of x. Let the declassification policy $\phi = \{\mathbb{Z}, \geq 0, < 0, \varnothing\}$. The wlp^P is $\{l = 0 \mapsto h = 0, l > 0 \mapsto h > 0, l < 0 \mapsto h < 0\}$ hence, the information released is $\phi' = \{\mathbb{Z}, \geq 0, \neq 0, \leq 0, > 0, 0, < 0, \varnothing\}$. If we compute the compressor then we obtain $\phi'' = \lambda X.\, \mathbb{Z}$, which means that every policy between ϕ' and ϕ'' is not able to protect the program.

4 The Geometry of Completeness Semantic Transformers

In this section, we introduce a completely symmetric construction for semantic transformers. The problem is: *Can we (minimally) transform the semantics in order to satisfy completeness?* The transformation is made in two steps: first we induce *completeness*, and then we force *monotonicity* by using standard results on function transformers in [7], since in some cases, the completeness transformation may generate not monotone functions. Recall that any function can be transformed to the closest (from below and from above) monotone function by considering the following basic transformers [7]:

$$\mathbb{M}^{\downarrow} \stackrel{\text{def}}{=} \lambda f.\, \lambda x.\, \bigwedge \{\, f(y) \mid y \geq x \,\} \qquad \mathbb{M}^{\uparrow} \stackrel{\text{def}}{=} \lambda f.\, \lambda x.\, \bigvee \{\, f(y) \mid y \leq x \,\}$$

Before introducing these transformers we have to understand what we mean by *minimally* transforming semantics. As usual we consider a lattice of functions where maps are point-wise ordered: $f \sqsubseteq g$ iff $\forall x \in C.\, f(x) \leq g(x)$. Hence, a minimal transformation of f finds the closest function, by reducing or increasing the images of f, wrt. a given property we want to hold for f (in this context, completeness). In abstract interpretation this corresponds to find the closest (viz., least abstraction or concretisation) of the semantics such that completeness holds for a given pair of abstractions. In the following, for simplicity, we consider the case of forward completeness.

4.1 Transforming Semantics for Inducing Forward Completeness

Moving Upwards. Let us consider first the case of increasing a given function $f : C \xrightarrow{\text{m}} C$ in order to induce completeness with respect to two fixed abstractions $\eta, \rho \in uco(C)$. We first observe that such a minimal transformation exists, namely the set $\{h : C \xrightarrow{\text{m}} C \mid f \sqsubseteq h,\ \rho \circ h \circ \eta = h \circ \eta\}$ has the minimal element. The following result proves that we can always minimally increase a given monotone function f in order to induce completeness.

Theorem 12. *The set* $\left\{ f : C \xrightarrow{m} C \,\middle|\, \rho \circ f \circ \eta = f \circ \eta \right\}$ *is an upper closure operator on* $\langle C \xrightarrow{m} C, \sqsubseteq \rangle$.

For any $f \in C \xrightarrow{\text{m}} C$ and $\eta, \rho \in uco(C)$ define

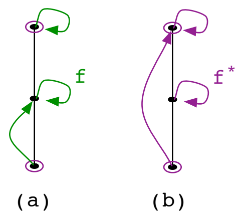
$$\mathbb{F}^{\uparrow}_{\eta,\rho} \stackrel{\text{def}}{=} \lambda f.\lambda x. \begin{cases} \rho \circ f(x) & \text{if } x \in \eta(C) \\ f(x) & \text{otherwise} \end{cases}$$

Lemma 13. *Let* $f : C \xrightarrow{m} C$*. Then*

$$\mathbb{F}^{\uparrow}_{\eta,\rho}(f) = \bigsqcap \left\{ h : C \longrightarrow C \,\middle|\, f \sqsubseteq h,\ \rho \circ h \circ \eta = h \circ \eta \right\}.$$

The function $\mathbb{F}^{\uparrow}_{\eta,\rho}(f)$ is not the one we look for since it may lack monotonicity. Next example shows that $\mathbb{F}^{\uparrow}_{\eta,\rho}(f)$ may not be monotone for some $f : C \xrightarrow{\text{m}} C$.

Example 14. Consider the pictures on the right. The (purple) circled points are those in ρ and the (green) arrows in the picture (a) represent f. The (purple) arrows in the picture (b) are those of the map obtained from f by applying $\mathbb{F}^{\uparrow}$ which is clearly not monotone.



The lack of monotonicity is due to the fact that, in order to minimally transform f, only the images of the elements in η are modified, leaving unchanged the images of all the other elements. Indeed monotonicity fails when we check it between the new image of one element in η and one outside. We need therefore to apply the transformer $\mathbb{M}^{\uparrow}$ for finding the best *monotone* transformation of f which is complete for the pair of abstractions $\eta, \rho \in uco(C)$. Define $\mathcal{F}^{\uparrow}_{\eta,\rho} \stackrel{\text{def}}{=} \lambda f.\, \mathbb{M}^{\uparrow} \circ \mathbb{F}^{\uparrow}_{\eta,\rho}(f)$.

Theorem 15. *Let* $f : C \xrightarrow{m} C$*.*

$$\mathcal{F}^{\uparrow}_{\eta,\rho}(f) = \bigsqcap \left\{ h : C \xrightarrow{m} C \,\middle|\, f \sqsubseteq h,\ \rho \circ h \circ \eta = h \circ \eta \right\}.$$

Moving Downwards. We consider the maximal approximation from below of a given $f : C \xrightarrow{\mathrm{m}} C$, making it complete This exists unique under some hypothesis.

Theorem 16. *The set* $\{f : C \xrightarrow{m} C \mid \rho \circ f \circ \eta = f \circ \eta\}$ *is a lower closure operator on* $\langle C \xrightarrow{m} C, \sqsubseteq\rangle$ *iff* ρ *is additive.*

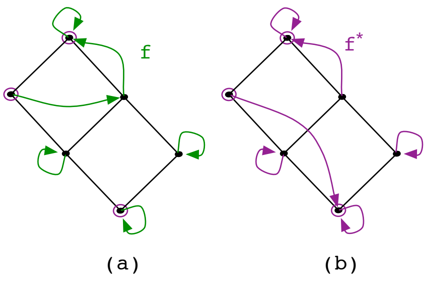
For any $f : C \xrightarrow{\mathrm{m}} C$ and additive closure $\rho \in uco(C)$, define:

$$\mathbb{F}^{\downarrow}_{\eta,\rho} \overset{\text{def}}{=} \lambda f.\lambda x. \begin{cases} \rho^{+} \circ f(x) & \text{if } x \in \eta(C) \\ f(x) & \text{otherwise} \end{cases}$$

Lemma 17. *Let* $f : C \xrightarrow{m} C$*, then*

$$\mathbb{F}^{\downarrow}_{\eta,\rho}(f) = \bigsqcup \left\{ \; h : C \longrightarrow C \;\middle|\; f \sqsupseteq h,\; \rho \circ h \circ \eta = h \circ \eta \; \right\}$$

Example 18. Consider the pictures on the right. The (purple) circled points are those in ρ and the (green) arrows on picture (a) are those of f. The (purple) arrows on the picture (b) are those of the map obtained from f by means of $\mathbb{F}^{\downarrow}$, and this map is clearly not monotone.



Again, the lack of monotonicity is due to the fact that, in sake of minimality, the transformers changes the image by f of only some elements, those of η. Again we apply the transformer $\mathbb{M}^{\downarrow}$ for finding the best *monotone* transformation of f. Next theorem shows that it is not necessary a fix-point transformation since the monotone transformer does not change the completeness of functions obtained by $\mathbb{F}^{\downarrow}$. Define $\mathcal{F}^{\downarrow}_{\eta,\rho} \overset{\text{def}}{=} \lambda f.\, \mathbb{M}^{\downarrow} \circ \mathbb{F}^{\downarrow}(f)$.

Theorem 19. *Let* $f : C \xrightarrow{m} C$*, then*

$$\mathcal{F}^{\downarrow}_{\eta,\rho}(f) = \bigsqcup \left\{ \; h : C \xrightarrow{m} C \;\middle|\; f \sqsupseteq h,\; \rho \circ h \circ \eta = h \circ \eta \; \right\}.$$

Next result tells us that the completeness transformers, moving in opposite directions in the lattice of functions, are indeed adjoint transformers whenever both exist, namely when the output abstraction ρ is additive, i.e. $\curlyvee(\rho) = \rho$.

Theorem 20. *If* $\rho \in uco(C)$ *is additive then* $(\mathbb{F}^{\uparrow}_{\eta,\rho})^{+} = \mathbb{F}^{\downarrow}_{\eta,\rho}$.

4.2 Transforming Semantics for Inducing Forward Incompleteness

Consider the two $\mathscr{F}$-completeness transformers for making functions complete for a given pair of domains, η on the input and ρ on the output. We proved that $\mathcal{F}^{\uparrow}_{\eta,\rho} \in uco(C \xrightarrow{\mathrm{m}} C)$ and $\mathcal{F}^{\downarrow}_{\eta,\rho} \in lco(C \xrightarrow{\mathrm{m}} C)$. We prove that their adjoint functions increase incompleteness. Given a function f, we look for the most distant function with the same complete transformation as f. In particular, when we consider $\mathcal{F}^{\uparrow}_{\eta,\rho}$, we look for the smallest function with the same transformation complete transformation as f which surely includes the maximal amount of incompleteness. A dual reasoning can be done for the other transformation following Janowitz's results.

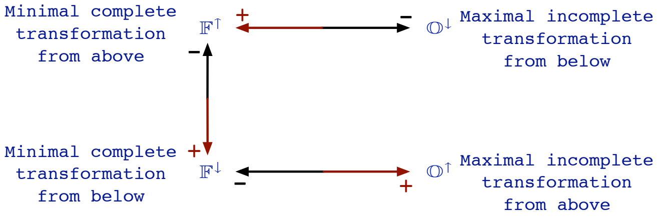

Minimal complete transformation from above
$\mathbb{F}^{\uparrow}$
+
−
$\mathbb{O}^{\downarrow}$
Maximal incomplete transformation from below
−
+
Minimal complete transformation from below
$\mathbb{F}^{\downarrow}$
−
+
$\mathbb{O}^{\uparrow}$
Maximal incomplete transformation from above

Fig. 2. Basic semantic transformers

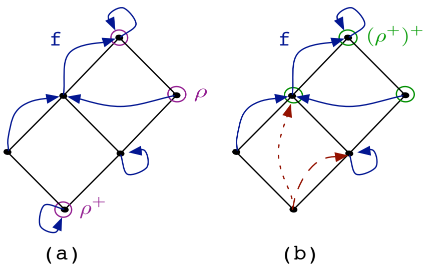
Moving Upwards. The corresponding right adjoint, when it exists (see Sec. 3), of $\mathcal{F}^{\downarrow}_{\eta,\rho}$ is $\mathcal{O}^{\uparrow}_{\eta,\rho}(f) \stackrel{\text{def}}{=} \bigsqcup\{g : C \xrightarrow{\text{m}} C \mid \mathcal{F}^{\downarrow}_{\eta,\rho}(g) = \mathcal{F}^{\downarrow}_{\eta,\rho}(f)\}$. Being $\mathcal{F}^{\downarrow}_{\eta,\rho}$ the composition of two function transformations, we study first the adjoint operation associated with $\mathbb{F}^{\downarrow}$: $\mathbb{O}^{\uparrow}_{\eta,\rho}(f) \stackrel{\text{def}}{=} \bigsqcup\{g : C \longrightarrow C \mid \mathbb{F}^{\downarrow}_{\eta,\rho}(g) = \mathbb{F}^{\downarrow}_{\eta,\rho}(f)\}$ and then prove that we can always apply the monotonicity transformer afterwards. As observed in Sect. 3, this adjoint operator may not always exist. Moreover, in the previous section we also observed that $\mathbb{F}^{\downarrow}$ may not always exist. Next theorem tells us that the incompleteness transformer exists when ρ^+ is join-uniform, which implicitly says that ρ^+ exists, namely that we also need ρ additive.

Theorem 21. *The transformer $\mathbb{O}^{\uparrow}_{\eta,\rho}(f) \in uco(C \longrightarrow C)$ iff ρ^+ exists and it is join-uniform. In this case*

$$\mathbb{O}^{\uparrow}_{\eta,\rho}(f)(x) = \begin{cases} (\rho^+)^+(f(x)) = \bigvee \{\, y \mid \rho^+(y) = \rho^+(f(x)) \,\} & \text{if } x \in \eta \\ f(x) & \text{otherwise} \end{cases}$$

Note that, also in this case, the transformer does not change all the elements, but only those in η. This implies that, also in this case, the function obtained by applying $\mathbb{O}^{\uparrow}$ is not necessarily monotone, even if applied to a monotone function.

Example 22. Consider $\eta = \rho$. In picture (a) the (purple) circled points are those in ρ (and in ρ^+) and the (blue) arrows corresponds to the function f. In picture (b) we have represented with (green) circled points the closure $(\rho^+)^+$ (which is an uco on the lifted order). We note that the transformation induced by $\mathbb{O}^{\uparrow}$ represented with a dotted line, is non monotone.



By applying $\mathbb{M}^{\downarrow}$ we obtain the monotone one represented with a dashed line.

The example above shows that by applying the transformer $\mathbb{M}^{\downarrow}$ we still obtain a monotone incomplete function. Next result proves that the transformer $\mathbb{M}^{\downarrow}$ does not change the class of complete functions, hence we can always make the monotonicity transformation if necessary, without having to reapply $\mathbb{O}^{\uparrow}$.

Theorem 23. *If* $f : C \xrightarrow{m} C$ *then* $\mathbb{F}^{\downarrow}_{\eta,\rho} \circ \mathbb{M}^{\downarrow} \circ \mathbb{O}^{\uparrow}_{\eta,\rho}(f) = \mathbb{F}^{\downarrow}_{\eta,\rho}$ *and* $\mathcal{O}^{\uparrow}_{\eta,\rho}(f) \sqsupseteq \mathbb{M}^{\downarrow} \circ \mathbb{O}^{\uparrow}_{\eta,\rho}(f)$.

At this point, we have that $\mathbb{O}^{\uparrow}$ is the adjoint of a transformer inducing completeness, hence intuitively it induces incompleteness. Clearly, we wonder if there are complete functions which are fix-point of $\mathbb{O}^{\uparrow}$. Next theorem proves that $\mathbb{O}^{\uparrow}$ does not always transform a complete function into an incomplete one. This is the case when no incomplete functions are available.

Theorem 24. *If* $f : C \xrightarrow{m} C$ *then* $\mathbb{O}^{\uparrow}_{\eta,\rho}(f)$ *is complete iff for each* $x \in C$ *we have* $\left\{\, y \,\middle|\, \rho^{+}(y) = \rho \circ f \circ \eta(x) \,\right\} = \{f \circ \eta(x)\}$.

In Fig. 3 (a) we show an example where the condition above holds also for non-trivial functions and closures where $\eta = \rho$. In the picture the closure is represented with (purple) circled points and the map with (green) arrows. This is a non-trivial complete function which is a fix-point of the transformer. In Fig. 3

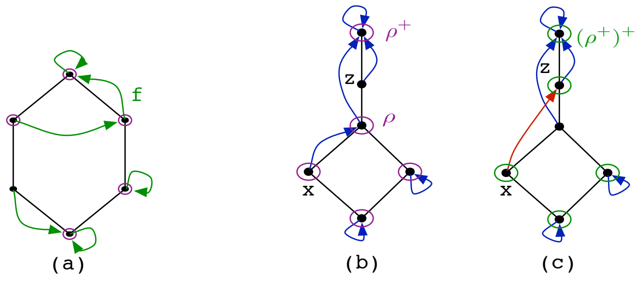


Fig. 3. Incompleteness transformations examples

(b) and (c) we show a case where a complete function is indeed transformed in an incomplete one by the transformer. Again, in Fig. 3 (b) the closure ρ (and ρ^{+}) is represented with (purple) circled points and the function f, for which the domain is complete, is represented with (blue) arrows. $\mathbb{O}^{\uparrow}_{\eta,\rho}(f)$ is in Fig. 3 (c).

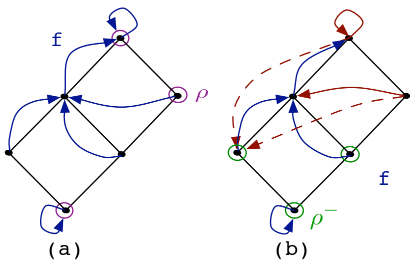
Moving Downwards. We close our construction of semantic transformers by characterising an incompleteness transformer associated with the left-adjoint of $\mathcal{F}^{\uparrow}_{\eta,\rho}$, in the sense of expansion: $\mathcal{O}^{\downarrow}_{\eta,\rho}(f) = \bigsqcap\{g : C \xrightarrow{m} C \mid \mathcal{F}^{\uparrow}_{\eta,\rho}(g) = \mathcal{F}^{\uparrow}_{\eta,\rho}(f)\}$ when it exists (see Sect. 3). As in the previous case, we study the following transformation first $\mathbb{O}^{\downarrow}_{\eta,\rho}(f) \overset{\text{def}}{=} \bigsqcap\{g : C \longrightarrow C \mid \mathbb{F}^{\uparrow}_{\eta,\rho}(g) = \mathbb{F}^{\uparrow}_{\eta,\rho}(f)\}$. While $\mathbb{F}^{\uparrow}_{\eta,\rho}$ always exists, $\mathbb{O}^{\downarrow}_{\eta,\rho}$ may not always exist. Next theorem proves that the incompleteness transformer exists when ρ^{-} exists, namely when ρ is meet-uniform.

Theorem 25. *$\mathbb{O}^{\downarrow}_{\eta,\rho}(f) \in lco(C \longrightarrow C)$ iff ρ is meet-uniform. In this case*

$$\mathbb{O}^{\downarrow}_{\eta,\rho}(f)(x) = \begin{cases} \rho^{-}(f(x)) = \bigwedge \{\, y \,|\, \rho(y) = \rho(f(x)) \,\} & \text{if } x \in \eta \\ f(x) & \text{otherwise} \end{cases}$$

Exactly as it happens in all previous cases, the transformer does not change all the elements, but only those in η. This implies that the function obtained by $\mathbb{O}^{\downarrow}$ may not be monotone, even if we start with a monotone function f.

Example 26. Consider the pictures on the right, where $\eta = \rho$. Here we have an example of transformation which returns a non monotone function. We see in picture (b), the closure ρ^{-} denoted with (green) circled points. The transformed map is the one represented in picture (b) with (red) dashed lines. Note that $\mathbb{M}^{\uparrow}(\mathbb{O}^{\downarrow}_{\eta,\rho}(f)) = f$.



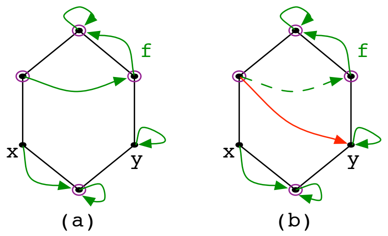
Theorem 27.
If $f : C \xrightarrow{m} C$ then $\mathbb{F}^{\uparrow}_{\eta,\rho} \circ \mathbb{M}^{\uparrow} \circ \mathbb{O}^{\downarrow}_{\eta,\rho}(f) = \mathbb{F}^{\uparrow}_{\eta,\rho}$ and $\mathcal{O}^{\downarrow}_{\eta,\rho}(f) \sqsubseteq \mathbb{M}^{\uparrow} \circ \mathbb{O}^{\downarrow}_{\eta,\rho}(f)$.

Also in this case, we consider the case when this trasformation induces incompleteness. Namely, we wonder when $\rho \circ \mathbb{O}^{\downarrow}_{\eta,\rho}(f) \circ \eta \neq \mathbb{O}^{\downarrow}_{\eta,\rho}(f) \circ \eta$.

Theorem 28. *If $f : C \xrightarrow{m} C$ then $\mathbb{O}^{\downarrow}_{\eta,\rho}(f)$ is complete iff for each $x \in C$ we have $\{\, y \,|\, \rho(y) = \rho \circ f \circ \eta(x) \,\} = \{f \circ \eta(x)\}$.*

An example of complete function which is left unchanged by $\mathbb{O}^{\downarrow}$ is the same shown before for $\mathbb{O}^{\uparrow}$ (Fig. 3 (a)). Next example shows, instead, a case where a complete function is indeed transformed into an incomplete one by $\mathbb{O}^{\downarrow}$.

Example 29. Consider the pictures on the right where $\eta = \rho$. Again the closure is represented with (purple) circled points. In the picture (a) we have the original function f, for which the domain is complete, as we can simply verify. In (b) we have the function that we obtain by applying the transformer $\mathbb{O}^{\downarrow}$ to f.



4.3 Transforming Semantics for Transforming Program Security

Transforming functions corresponds to transform semantics [10]. In the following we consider the meaning of transformed semantics in language-based security.

Inducing Completeness. Assume η be disjunctive, i.e. $\curlyvee(\eta) = \eta$. We consider $\mathbb{F}^{\uparrow}_{\mathcal{H}^{\phi}_{\eta},\mathcal{H}_{\rho}}(wlp^{P}_{\eta})$ and $\mathbb{F}^{\downarrow}_{\mathcal{H}^{\phi}_{\eta},\mathcal{H}_{\rho}}(wlp^{P}_{\eta})$, where $\mathcal{H}^{\phi^{+}}_{\eta} = \lambda\langle X^{\mathtt{H}}, X^{\mathtt{L}}\rangle.\, \langle \phi^{+}(X^{\mathtt{H}}), \eta^{+}(X^{\mathtt{L}})\rangle$.

The transformations above, show that whenever we have $wlp_\eta^P(X) = \langle H, \eta(L)\rangle$, and $X \in \mathcal{H}_\rho$ then $\mathbb{F}^\uparrow_{\mathcal{H}_\eta^\phi, \mathcal{H}_\rho}(wlp_\eta^P)(X) = \langle \phi(H), \eta(L)\rangle$ by idempotence of η, and $\mathbb{F}^\downarrow_{\mathcal{H}_\eta^\phi, \mathcal{H}_\rho}(wlp_\eta^P)(X) = \langle \phi^+(H), \eta(L)\rangle$ by Th. 1(1). This result tells us that, while by using the core we can transform the output observation ρ for characterizing attackers, we cannot transform the input observation η.

Example 30. Let $P \overset{\text{def}}{=}$ **while** $(h > 0)$ **do** $(h := h - 1; l := h)$ **endw**. Suppose the attacker can observe the identity. The *wlp* is $\{l = 0 \mapsto \; h \geq 0, l \neq 0 \mapsto h = 0\}$. This program is insecure for the policy $\phi = \lambda x. \mathbb{Z}$, which says that nothing has to flow. The secure semantic transformation is $\mathbb{F}^\uparrow_{\mathcal{H}_{id}^\phi, \mathcal{H}_{id}}(wlp_{id}^P)$: $\{l = 0 \mapsto \; h \in \mathbb{Z}, l \neq 0 \mapsto h \in \mathbb{Z}\}$ which, for example, is the semantics of the transformed program $P' \overset{\text{def}}{=} l_1 := l; P; l := l_1$.

Inducing Incompleteness. Let us consider the incompleteness transformers $\mathbb{O}^\uparrow_{\mathcal{H}_\eta^\phi, \mathcal{H}_\rho}(wlp_\eta^P)$ and $\mathbb{O}^\downarrow_{\mathcal{H}_\eta^\phi, \mathcal{H}_\rho}(wlp_\eta^P)$ where, when the necessary conditions on η and ϕ hold, $\mathcal{H}_\eta^{\phi^{++}}(X) = \langle \phi^{++}(X^{\mathtt{H}}), \eta^{++}(X^{\mathtt{L}})\rangle$ and $\mathcal{H}_\eta^{\phi^-}(X) = \langle \phi^-(X^{\mathtt{H}}), \eta^-(X^{\mathtt{L}})\rangle$. The problem is that the completeness equation with ϕ holds if ϕ is partitioning [1]. This is a problem since it implies that ϕ is not meet-uniform and ϕ^+ cannot be join-uniform. Hence we don't have an optimal transformation of wlp_η^P but rather only maximal incomplete transformations. This is interpreted by noting that a semantic transformer corresponds to an active attacker that wants to exploit its *activity* for disclosing more private information. A program that does not reveal to an active attacker more than what is revealed to a passive one is called *robust* [23]. Therefore there is no best active attacker that can extract all about private data. An attacker can only decide what it wants to disclose and consequently actively transform the code.

Example 31. Let $P \overset{\text{def}}{=} h := h \; mod \; 2;$ **if** $h = 0$ **then** $l := 0$ **else** $l := 1$ and $\rho = \eta = id$, $\phi = Par$. wlp_{id}^P is $\{l = 0 \mapsto h \in 2\mathbb{Z}, l = 1 \mapsto h \in 2\mathbb{Z}+1\}$ namely we disclose parity of h. Suppose the attacker wants to distinguish in addition if h is 0 or not. Then we consider the partitioning closure $\sigma \overset{\text{def}}{=} \Upsilon(\{\{0\}, 2\mathbb{Z} \smallsetminus \{0\}, 2\mathbb{Z} + 1\})$. For what we said above, we don't have a best transformation allowing to observe 0, hence the attacker has to choose to disclose weaker information about h, for example $\{0, 2\}$ instead of 0. We can consider the semantics: $\{l = 0 \mapsto h \in \{0, 2\}, l = 1 \mapsto h \in 2\mathbb{Z}+1\}$ approximating $\mathbb{O}^\downarrow_{\mathcal{H}_{id}^\sigma, \mathcal{H}_{id}}(wlp_{id}^P)$ which, for example, is the semantics of **if** $h \leq 2 \vee h \; mod \; 2 = 1$ **then** P **else** $l := 2$

5 Discussion

We proved that standard abstract interpretation methods, based on Galois connections, can provide an adequate model for reasoning about transformations of both abstract domains and semantics. While the abstract domain side is more traditional in static program analysis, in particular in the field of abstraction design, the semantic side is completely new. In this paper we proved that a completely symmetric construction holds for both semantic and domain transformers, sharing the same geometric structure which is based on the lifting of

Galois connections higher order, from the objects of computation to the space of abstract domains and predicate transformers. This shows that abstract interpretation, as originally developed in [6], may have a universal validity not only for approximating semantics but also on reasoning about its own methods. The key aspect in this construction is completeness, which is the driving force for transforming domains and semantics to achieve a given precision degree. We used language-based security as an application ground for interpreting our transformations, but the validity of these results are general. For instance possible applications of the basic semantic transformers for achieving maximal incompleteness are in code obfuscation. In this case an obfuscated program fighting against an attacker which performs static analysis driven reverse engineering can be viewed as the maximal incomplete transformation of the program with respect to the abstractions used by the analyser. Similarly minimal complete transformations can be used in abstract model checking for isolating temporal sub-logics which are complete for a given abstract system to analyse.

Acknowledgements. Part of the material developed in this paper was conceived with Francesco Ranzato and Elisa Quintarelli. We are grateful to the endless discussions we had together during the last decade, discussions that helped us in better understanding the beauty of abstract interpretation theory.

References

1. Banerjee, A., Giacobazzi, R., Mastroeni, I.: What you lose is what you leak: Information leakage in declassifivation policies. In: Proc. of the 23th Internat. Symp. on Mathematical Foundations of Programming Semantics MFPS 2007. ENTCS, vol. 1514. Elsevier, Amsterdam (2007)
2. Blyth, T.S., Janowitz, M.F.: Residuation theory. Pergamon Press, Oxford (1972)
3. Cortesi, A., Filé, G., Giacobazzi, R., Palamidessi, C., Ranzato, F.: Complementation in abstract interpretation. ACM Trans. Program. Lang. Syst. 19(1), 7–47 (1997)
4. Cousot, P.: The calculational design of a generic abstract interpreter. In: Broy, M., Steinbrüggen, R. (eds.) Calculational System Design. NATO ASI Series F. IOS Press, Amsterdam (1999)
5. Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theor. Comput. Sci. 277(1-2), 47–103 (2002)
6. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of Conf. Record of the 4th ACM Symp. on Principles of Programming Languages (POPL 1977), pp. 238–252. ACM Press, New York (1977)
7. Cousot, P., Cousot, R.: A constructive characterization of the lattices of all retractions, preclosure, quasi-closure and closure operators on a complete lattice. Portug. Math. 38(2), 185–198 (1979)
8. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proc. of Conf. Record of the 6th ACM Symp. on Principles of Programming Languages (POPL 1979), pp. 269–282. ACM Press, New York (1979)

9. Cousot, P., Cousot, R.: Comparing the Galois connection and widening/narrowing approaches to abstract interpretation (invited paper). In: Bruynooghe, M., Wirsing, M. (eds.) Proc. of the 4th Internat. Symp. on Programming Language Implementation and Logic Programming (PLILP 1992). LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
10. Cousot, P., Cousot, R.: Systematic design of program transformation frameworks by abstract interpretation. In: Proc. of Conf. Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp. 178–190. ACM Press, New York (2002)
11. Filé, G., Giacobazzi, R., Ranzato, F.: A unifying view of abstract domain design. ACM Comput. Surv. 28(2), 333–336 (1996)
12. Giacobazzi, R., Mastroeni, I.: Domain compression for complete abstractions. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 146–160. Springer, Heidelberg (2002)
13. Giacobazzi, R., Mastroeni, I.: Abstract non-interference: Parameterizing non-interference by abstract interpretation. In: Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2004), pp. 186–197. ACM-Press, New York (2004)
14. Giacobazzi, R., Mastroeni, I.: Adjoining declassification and attack models by abstract interpretation. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 295–310. Springer, Heidelberg (2005)
15. Giacobazzi, R., Quintarelli, E.: Incompleteness, counterexamples and refinements in abstract model-checking. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 356–373. Springer, Heidelberg (2001)
16. Giacobazzi, R., Ranzato, F.: Refining and compressing abstract domains. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) ICALP 1997. LNCS, vol. 1256, pp. 771–781. Springer, Heidelberg (1997)
17. Giacobazzi, R., Ranzato, F.: Optimal domains for disjunctive abstract interpretation. Sci. Comput. Program 32(1-3), 177–210 (1998)
18. Giacobazzi, R., Ranzato, F.: Uniform closures: order-theoretically reconstructing logic program semantics and abstract domain refinements. Inform. and Comput. 145(2), 153–190 (1998)
19. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. J. of the ACM. 47(2), 361–416 (2000)
20. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract domains condensing. ACM Transactions on Computational Logic (ACM-TOCL) 6(1), 33–60 (2005)
21. Giacobazzi, R., Scozzari, F.: A logical model for relational abstract domains. ACM Trans. Program. Lang. Syst. 20(5), 1067–1109 (1998)
22. Janowitz, M.F.: Residuated closure operators. Portug. Math. 26(2), 221–252 (1967)
23. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. on selected ares in communications 21(1), 5–19 (2003)
24. Ward, M.: The closure operators of a lattice. Ann. Math. 43(2), 191–196 (1942)

Reflections on the Role of Static Analysis in Cooperative Bug Isolation*

Ben Liblit

Computer Sciences Department
University of Wisconsin–Madison
liblit@cs.wisc.edu

Abstract. Cooperative Bug Isolation (CBI) is a feedback-directed approach to improving software quality. Developers provide instrumented applications to the general public, and then use statistical methods to mine returned data for information about the root causes of failure. Thus, users and developers form a feedback loop of continuous software improvement. Given CBI's focus on statistical methods and dynamic data collection, it is not clear how static program analysis can most profitably be employed. We discuss current uses of static analysis during CBI instrumentation and failure modeling. We propose novel ways in which static analysis could be applied at various points along the CBI feedback loop, from fairly concrete low-level optimization opportunities to hybrid failure-modeling approaches that may cut across current static/dynamic/statistical boundaries.

1 Introduction

A complete Cooperative Bug Isolation (CBI) system constitutes a feedback loop between developers and users. Developers provide software to users, and users respond with data about that software's behavior in the deployed environment. Developers then use this data to improve the software in future releases, guided largely by sophisticated statistical models of program misbehavior. The goal is not to produce perfect first releases, but rather to improve software continuously over time guided by the needs of real user communities [1]. It is non-obvious how formal static analysis should interact with CBI's dynamic/statistical approach to software quality. This paper explores how static analysis is currently used within CBI, and how it could most beneficially be used in the future.

1.1 Overview of Cooperative Bug Isolation

We start with a conceptual overview of CBI's feedback loop in Fig. 1. The process begins at the top left, with program source and a small set of configuration choices (made by the developer) that will steer a CBI instrumenting compiler. Configuration choices

* Supported in part by AFOSR Grant FA9550-07-1-0210 and NSF Grants CCF-0621487, CCF-0701957, and CNS-0720565. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of AFOSR, NSF, or other institutions.

© Springer-Verlag Berlin Heidelberg 2008

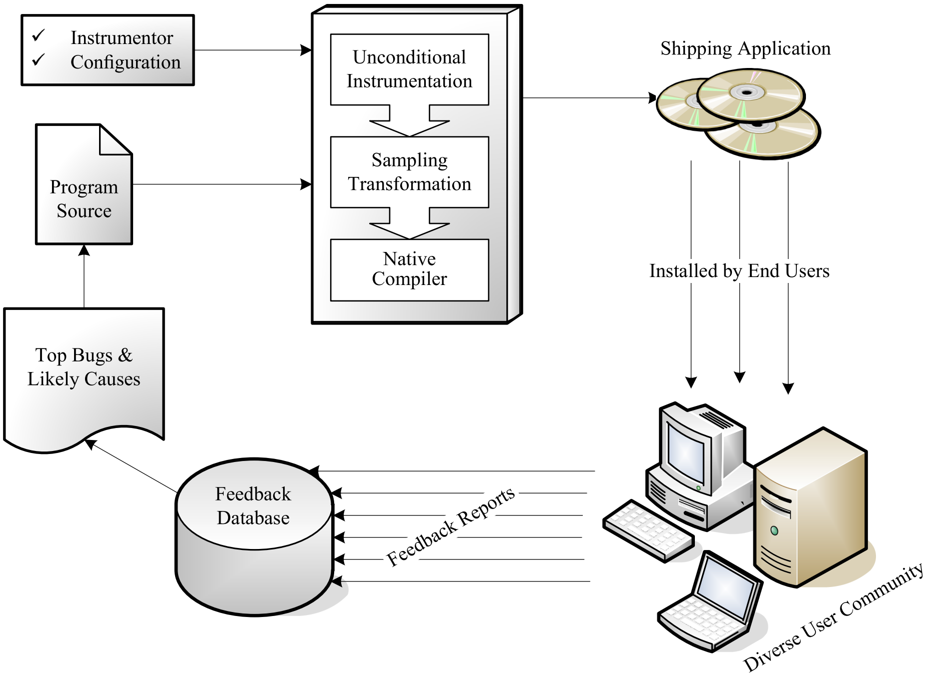


Fig. 1. Conceptual overview of Cooperative Bug Isolation system

are simple and few in number, while program source is absolutely free of any manual, CBI-specific annotation. This minimizes the up-front cost to developers wishing to adopt CBI.

The boxed tool chain at the top center of Fig. 1 represents a CBI instrumenting compiler. This appears to the developer to be a standard compiler augmented with a few additional high-level configuration options, and therefore is easily incorporated into existing build environments. Internally, CBI-instrumented compilation consists of three distinct steps: (1) insertion of unconditional instrumentation; (2) transformation of instrumentation to be sampled instead of unconditional; and (3) native compilation to binaries ready for distribution. The third step is treated as a black box: we accept whatever the native compiler gives us. But each of the first two steps represents a key point at which static analysis could be used to good effect. We review unconditional instrumentation in Sect. 2, with a discussion of the ways in which static analysis is currently used and could potentially be used in the future. Section 3 recaps the sampling transformation, and again considers the current and potential roles for static analysis.

Instrumented compilation yields executable binaries (both applications and shared libraries) capable of monitoring and reporting aspects of their own behavior at run time. We make this instrumented software available to the general public: the “diverse user community” in the lower right of Fig. 1. Each run of an instrumented program produces one feedback report with information about run time behaviors instrumented during compilation. Each feedback report also includes a binary label marking that run as good (successful) or bad (failed). The challenges at this stage largely fall under the broad umbrella of "computer systems": transmitting feedback data over the network, warehousing it in databases, keeping user data secure throughout its lifetime, and so on. Static program analysis plays no role here.

As runs accumulate, certain trends emerge. If we put all successful runs in one pile and all failed runs in another, we can look for instrumented behaviors that vary between the two piles. In particular, we can direct developers' attention toward suspicious run time behaviors that are strongly associated with failure. Absolute assignment of blame ("the program crashes if and only if `x` is negative on line 140") is impossible due to the many sources of uncertainty in our feedback data. Instead, we have developed a variety of *statistical debugging* techniques to identify informative trends in the difference between successful and failed runs. Statistical debugging is another key point in the feedback loop where static program analysis can play a major role, although many open questions remain. In Sect. 4 we review selected statistical debugging models developed both by my collaborators and by others. We consider the relatively limited use of static program analysis in statistical debugging work thus far, and suggest future directions in which program analysis could play a more prominent role.

Certain recurring challenges span all of the contexts in which we consider adding static analysis to CBI. Section 5 vents some steam about problems that have put many attractive static analyses out of CBI's reach. For the stout-hearted reader who is not so offended as to discard the paper following this rant, Sect. 6 concludes.

2 Unconditional Instrumentation

The first step in compiling an application for use with CBI is to inject extra monitoring code at selected program points of interest. We call this code *unconditional instrumentation* to contrast it with the sampled instrumentation that results from later transformations (see Sect. 3). Unconditional instrumentation casts a broad net across the program, adding code to record a wide variety of run time behaviors that could potentially be of interest later when tracking down a bug. This is not the place to be stingy: we regularly add many hundreds of thousands of monitoring points to medium-sized programs.

Obviously we do not expect the programmer to insert this code by hand. Rather, the developer selects among a small collection of broad *instrumentation schemes*. Each scheme matches specific fragments of program syntax at specific locations in code. We call such matches *instrumentation sites*. For example, the *branches scheme* places one instrumentation site at each `if` statement, while the *returns scheme* creates one site at each function call. Possible behaviors at each site are strictly partitioned into a small set of *predicates* on program state. For example, a branch site tests two predicates: the `if` condition is either true or false. We create one global *predicate counter* for each predicate, which records how often that predicate was true during a run. When execution reaches the location of an instrumentation site, a single *observation* is made: exactly one of the predicates at the site must be true, and the injected code makes this determination and increments the corresponding predicate counter. The feedback report for an entire run, then, consists primarily of the final counter values for all predicates.

`sampler-cc` is our CBI instrumenting compiler for C. It is not the only instrumentor that has been created [2,3,4], but it is the most mature and widely-used. `sampler-cc` currently offers seven instrumentation schemes; developers may activate as many or as few as desired [5]. Some schemes, such as the branches scheme, are quite broad-based and stand a decent chance of detecting something of interest for a wide variety of coding mistakes. Other schemes are more specialized, such as the *g-object-unref scheme* which focuses on reference count mismanagement bugs in a specific widely-used open source library. Specialized schemes are less likely to trigger for any given bug, but if they do correlate strongly with failure, the guidance they offer the developer is more specific and therefore more useful. We generally recommend a mixture of broad-spectrum and narrowly-focused schemes.

2.1 Static Analysis as Currently Used

`sampler-cc` performs almost no interesting static analysis during the unconditional instrumentation stage. This is a somewhat embarrassing admission. Unconditional instrumentation relies on the popular CIL C front end for parsing, type checking, and name resolution [6], and arguably that constitutes program analysis of a sort. But beyond these standard front-end tasks, initial instrumentation is essentially a matter of local pattern-matching against syntactic structures (e.g., finding every conditional branch) and insertion of appropriate monitoring code (e.g., to count how often the branch condition is true versus false). We rarely look at more than one small abstract syntax tree fragment at a time.

The one exception is a relatively recent feature that drops instrumentation sites that examine the values of uninitialized variables. The motivation is not that looking at such values is dangerous or forbidden. After all, this is C: everything is dangerous, and nothing is forbidden. Rather, we find that developers cannot easily make sense of instrumented predicates that involve uninitialized variables. Therefore, considering such predicates when hunting for bugs is not ultimately useful, even if some may be strong failure predictors. So `sampler-cc` uses an intraprocedural forward dataflow analysis to identify and avoid definitely-uninitialized variables.

2.2 Static Analysis Potential

Historically, we have claimed that deep analysis of buggy C programs is foolhardy. How many static analyses give truthful results for C programs that overrun buffers or read uninitialized memory? In a world of wild pointers and corrupted heaps, what points-to analysis can possibly be trusted? The execution semantics of such programs are very different from the formal semantics assumed by most analyses, which means that static analyses can no longer offer guarantees across all possible runs.

Furthermore, many of the facts that static analysis could offer are much easier to derive empirically by examination of dynamic feedback reports. Ernst's Daikon work shows that empirical invariant discovery can be quite effective [7]. Why try to *prove* that `z` must always be zero on line 490 when we can simply check whether it was ever *observed* to be nonzero across a hundred thousand user runs? Of course the later does not provide any guarantees, but given the loose semantics of buggy C programs, it's not clear that any static analysis could offer guarantees either.

However, this may be prematurely dismissive. Static analysis could go a long way toward streamlining instrumentation by eliminating redundancy. A static proof that one predicate implies another means that the later need not be monitored at run time. For example:

```
1  const int result = fetch();
2  if (result) ...
```

The returns instrumentation scheme will check whether `result` is negative, zero, or positive on line 1. The branches scheme will check whether the conditional on line 2 is true or false. Is later redundant with respect to the former? Probably. If we assume that no other thread can change `result` between lines 1 and 2 (including by storing across a corrupted pointer), then instrumenting the branch is redundant. Should we make this assumption? Arguably, yes. It may cause us to miss certain bugs if we are wrong. But CBI never promised perfection. Reducing the instrumentation load by eliminating redundant or invariant predicates would have several benefits:

1. Less instrumentation means less code, for a smaller executable footprint on installation media, hard drives, and memory.
2. If sampling (discussed below) is not changed, then less instrumentation means more streamlined code and therefore better performance.
3. Conversely, if run time performance is held at a constant level, then less time wasted on uninteresting instrumentation sites allows more intensive sampling of other code that may be more informative.
4. Although statistical analysis of feedback reports can discover likely invariants and redundancies in observed data, this is not free. Advanced statistical debugging algorithms can have difficulty scaling up to massive datasets, and eliminating junk instrumentation earlier leaves the statistical methods with smaller problems to solve.

3 From Unconditional Instrumentation to Sampling

Here we detail CBI's instrumentation sampling transformation, which sacrifices feedback completeness for privacy and performance. We review static analyses currently used during the sampling transformation, and consider possible deeper analyses that could be employed in the future.

3.1 The CBI Sampling Transformation

Complete monitoring of all instrumentation predicates may be impractical or undesirable for reasons of performance or user privacy. We have developed a generic instrumentation sampling transformation to address these concerns [8]. CBI's sampling transformation is a statistically rigorous variant on a performance profiling transformation developed by Arnold and Ryder [9]. The sampling transformation reduces instrumentation overhead while maintaining a very strict statistical fairness guarantee: the behaviors observed and tallied in site counters represent a sparse but statistically unbiased random sample with respect to the complete (but unobserved) dynamic behavior of the program. This fairness guarantee is necessary to ensure that the statistical debugging algorithms to be applied later have a sound mathematical footing.

At run time, each thread in an instrumented application maintains a countdown that represents the number of instrumentation opportunities that should be skipped before the next observation is taken. The countdown is set randomly using a geometric distribution, which is mathematically equivalent to counting how many times a tail-biased coin comes up tails before the next head is seen. A geometric distribution with mean 100 corresponds to counting tails while tossing a coin that has a 1/100 chance of coming up heads. This countdown yields a statistically fair random sample of about 1/100 of complete program behavior, as though the biased coin were being tossed at each instrumentation site.

The sampling transformation leverages this countdown by avoiding most instrumentation until an observation is imminent. The transformation begins by splitting each function's control flow graph into a collection of single-entry acyclic subgraphs. Doing this optimally is NP-hard [10], but placing acyclic subgraph entry points at loop back edges and the tops of functions works well in practice. An acyclic graph contains only a finite number of paths, each of which is of finite length. Therefore, there is a finite maximum number of instrumentation sites that could be crossed along any single execution through each acyclic subgraph. We call this maximum instrumentation site count the *threshold weight* of a given subgraph.

An instrumenting compiler now clones each acyclic subgraph. In the *fast clone*, we replace each instrumentation site with a simple decrement of the global next-sample countdown. In the *slow clone*, we decrement the countdown and check whether it has reached zero. If the countdown is zero, then we make a single observation at the current instrumentation site and reset the counter to a new geometrically-distributed random number. Entry into the fast and slow clones is guarded by a conditional branch that checks whether the global countdown is below the subgraph's threshold weight. If the countdown is larger than this threshold, then the decrements cannot possibly drive it to zero on this pass through the subgraph, and so the branch selects the fast clone. If the countdown is smaller than the threshold, then a sample might be imminent, and the instrumented slow clone is selected instead. If sampling is sparse (1/100 or 1/1000 is typical), then execution will usually proceed in the fast clone of each subgraph, switching to the slow clone only occasionally when a sampled observation is about to be made. In this way we improve performance by exploiting periods between samples as the fast, common case.

Figure 2 shows an example of an acyclic control flow subgraph after cloning and the insertion of the countdown threshold check. This subgraph has a threshold of four, because there is one path through the subgraph that crosses four instrumentation sites.

3.2 Static Analysis as Currently Used

Some static analyses are implied by the description of the sampling transformation given above. We require control flow graphs for each function, with loop back edges identified. The threshold weight of each single-entry acyclic subgraph is computed in a simple bottom-up pass. While this all constitutes analysis, it has nothing beyond the basics that any reasonable compiler would provide.

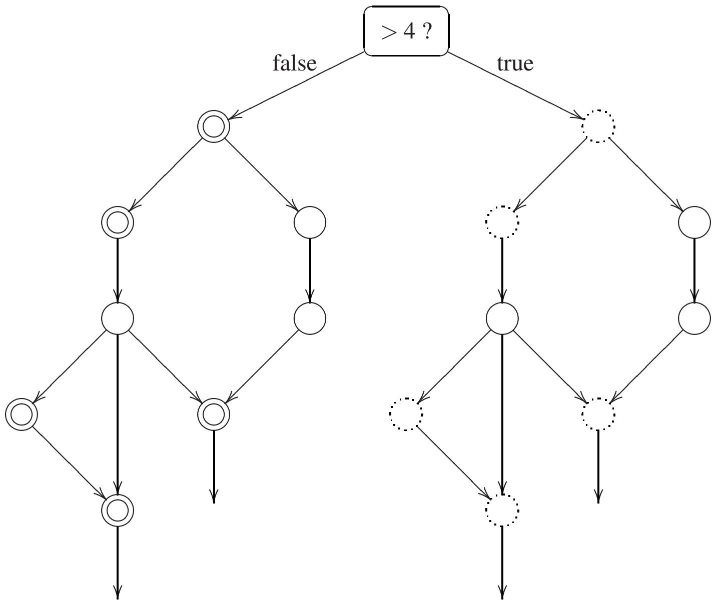


Fig. 2. Example of instrumented code layout. The slow clone is on the left; double-outlined nodes contain countdown decrements and instrumentation site code. The fast clone is on the right; dotted-outline nodes contain only countdown decrements. Single-outlined nodes contain no instrumentation sites.

Several other minor optimizations can be applied within each acyclic subgraph or instrumented function. For example, we add a static branch prediction hints to advise the native code generator that most threshold checks will choose the fast clone. Acyclic subgraphs containing zero or one instrumentation site require no cloning or threshold check at all. The global next-sample countdown is cached in a local variable while an instrumented function executes; this helps the native compiler coalesce decrements into fast register operations for a significant performance boost.

The only analysis that spans procedure boundaries is our identification of weightless functions. We define a *weightless function* as one that contains no instrumentation sites and that only calls other weightless functions. Weightless functions allow several optimizations in global countdown management, so we identify these using a fix-point computation over the call graph with conservative treatment of calls across pointers or that leave the current compilation unit. (A points-to analysis is offered as an option for resolving indirect calls. However, this is considered experimental and not recommended for production use due to its insufficiently-conservative treatment of separate compilation.)

3.3 Static Analysis Potential

Optimization of sampled instrumentation is primitive, with only very modest attempts to analyze beyond the boundary of a single function or a single acyclic subgraph. One could certainly do better, such as by optimizing across procedure boundaries or by restructuring the fast clones for even greater speed. We propose two analysis-driven optimizations in detail as examples of the sort of improvements that could be made.

Bounded-Weight Function Analysis. With the exception of weightless functions, we currently treat each called function as a black box that might contain arbitrarily-many instrumentation sites. Thus, the next-sample countdown may change arbitrarily across any non-weightless function call. This requires splitting acyclic subgraphs at function calls, which in turn makes the subgraphs smaller. Smaller acyclic subgraphs require more frequent threshold checks, harming performance.

Suppose instead we were to identify a maximum threshold weight for an entire function. For some functions this may not be bounded, but for many (especially small, loop-free leaf functions) a finite bound will exist. This information can be exploited by the caller to compute its own acyclic subgraph thresholds, since a call to a function with threshold weight n can only reduce the next-sample countdown by at most n from the perspective of the caller. Weightless functions are simply a special case of the more general class of bounded-weight functions.

Path Balancing. When the fast clone consists of simple, straight-line code, a native compiler may be able to coalesce multiple countdown decrements into a single larger adjustment. For example, `gcc` performs this optimization provided that the countdown is cached in a local variable per Sect. 3.2. However, decrement coalescing cannot extend across branches, because the multiple forward paths may contain different numbers of instrumentation sites and therefore require different net adjustments to the countdown.

Path balancing generalizes decrement coalescing to arbitrary acyclic subgraphs. The key is to ensure that all forward paths through an acyclic subgraph cross the same number of instrumentation sites. Imbalances occur at branches. When a control flow graph node has multiple successor paths with different weights, extra "dummy" sites are added to the start of those successor paths that have fewer "real" sites than their siblings, thereby creating balance. When all branches in a subgraph are balanced, the entire subgraph is balanced as well.

Figure 3a gives an example of an acyclic subgraph before balancing. Nodes with instrumentation sites have dotted outlines. Notes are lettered for ease of reference, and the number in each node gives the maximum weight of all paths forward from that node. The entire subgraph has threshold weight 2 but individual paths cross 0 (*abe*), 1 (*adh*), or 2 (*abcfg*, *abcfh*) sites. Branch nodes *a*, *b*, and *f* may require balancing. Branch *a* does have imbalanced successors: one dummy site must be added on the *ad* edge. Branch *b* is also imbalanced: two dummy sites must be added on the *be* edge. Branch *f* is already balanced: both successors already have matching weights.

Figure 3b shows the same acyclic subgraph after balancing. Three unlettered dummy sites have been added. The threshold weight for the entire subgraph (2) is now the exact number of sites crossed on each of the four paths through the subgraph starting from entry node *a*.

Balancing is not an optimization in and of itself. Rather, it actually adds instrumentation in the form of dummy sites. However, once a site is balanced, we can optimize the code as follows. Just before the first node of the fast clone, decrement the next-sample

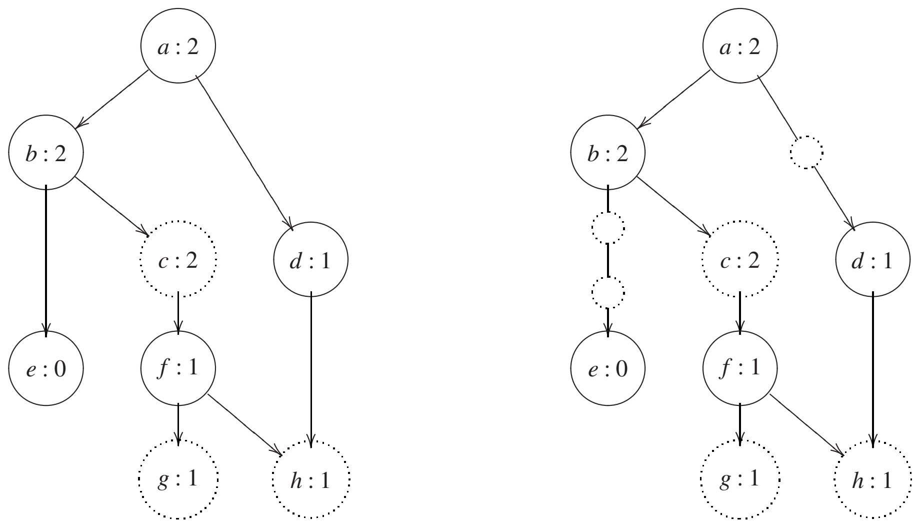


Fig. 3. Example of path balancing

countdown by the threshold weight of the entire subgraph (for example, "`countdown -= 2`" just before node *a* in Fig. 3b). This decrement accounts for exactly the number of unary decrements that would have occurred in this subgraph. Elsewhere in the fast clone, wherever a real or dummy instrumentation site would have appeared, do nothing. The decrements have already been accounted for and there is no other work to do.

The slow clone must decrement and check the countdown at each instrumentation site as before, because on the slow clone we do need to know exactly when a site should be sampled. Furthermore, even dummy sites must decrement the countdown and reset it if it reaches zero. This requirement ensures that both the fast and slow clones behave the same with respect to counting down to the next sample, at the expense of making the slow clone even slower. Also, adding dummy instrumentation sites means that the countdown will need to be reset more often, so a slow random number generator will be more of a liability here.

In total, path balancing makes the fast clone faster and the slow clone slower. The idea for the path balancing algorithm arose in discussions between the author and Cormac Flanagan, but has not yet been implemented or evaluated. We offer it here as an example of leveraging the regular structure of sampled instrumentation code using an intraprocedural analysis of quite modest complexity.

4 Statistical Debugging

Some instrumented behaviors may always occur or may never occur; these are invariants in practice (and possibly in theory). Most behaviors vary from run to run. If variation in some instrumented predicate correlates with failure of runs, then we call that predicate a *bug predictor*. The correlation may be imperfect: nondeterministic bugs can allow apparent success even in runs that ought to have failed. Sparse sampling means that even a completely deterministic failure predictor will not be observed on most runs where it does occur. For this reason, we must look for broad statistical trends in program (mis)behavior across large numbers of runs. A single run tells us virtually nothing, but because the sampling transformation is statistically unbiased, trends over many runs can guide developers to the root causes of recurrent problems.

Statistical debugging refers to the task of finding bug-predictive behaviors among the feedback data collected from large numbers of instrumented runs. Members of the machine learning community have expressed considerable interest in this problem, which can be seen as an unusual example of feature selection (finding bad behaviors) or clustering (grouping failures by the bug that caused them). Statistical models considered, either by me and my collaborators or by completely independent groups, include regularized logistic regression [8,11], probability density function comparison [12], likelihood ratio testing [13,14], iterative bipartite graph voting [15,16] three-valued logic [17] support vector machines [18], random forests [18], Delta Latent Dirichlet Allocation [19], and quite possibly others of which I am shamefully unaware. We refrain from reviewing the details of any of these algorithms here; the interested reader may read the original papers, perhaps with a statistics textbook or colleague nearby for guidance. The approaches vary in their ability to deal with multiple bugs, non-deterministic failures, sparsely sampled data, extremely large datasets, and other qualities. Most share a similar structure of analysis output: a ranked list of instrumented program behaviors that have been identified as bug predictors. Such a list can be presented to a developer to guide further triage, diagnosis, and remediation.

Given the messy, incomplete nature of sampled feedback data, one might think that formal static program analysis has little to contribute to statistical debugging. This is incorrect. Some of CBI's most interesting statistical analysis work takes place at this late stage, *after* the core statistical models have been built and used to identify bug predictors. Indeed, the analysis methods used here far outstrip those found in CBI instrumenting compilers, both in terms of their current sophistication and in their future potential.

4.1 Static Analysis as Currently Used

When hunting down a bug, a ranked list of bug predictors is a good start but it is not the complete story. Developers must still understand why the highlighted misbehaviors can lead to failure, and this is not always easy. Several static program analyses have been used to place bug predictors back into the context of the source program and help developers understand how they relate to failure.

One common goal is to stitch isolated bug-predictive program points together into extended failure paths. Developers can then walk through a doomed run (or an approximate reconstruction thereof) step by step to see where things fell apart. My own early attempts at this [20] have been bested by subsequent work by Lal et al. [21]. Lal's approach uses weighted pushdown systems, a powerful generic formalism for expressing context-sensitive interprocedural dataflow analyses [22,23]. Propelled by this engine, Lal's analysis reconstructs paths that proceed from program entry through high-ranked bug predictors to a point of failure. Paths obey feasibility constraints imposed by any of a variety of dataflow analyses. Jiang and Su build partial faulty-path segments using an efficient control-flow graph traversal with backtracking [18]. The search is a static analysis, but is heuristically guided by using CBI feedback data to guess likely execution paths at branches.

Static analysis also plays a role in comparing the quality of ranked bug predictor lists. Cleve and Zeller [24] propose a quantitative approach that measures the distance between the code blamed by some tool and the actual location of the bug. Distances are measured in the program dependence graph (PDG), following a model by Renieris and Reiss [25]. This metric is intended to simulate an idealized programmer who first examines code blamed by the tool, then proceeds outward across PDG edges until the true flaw is found. Numerous other researches (myself included) have adopted this PDG-distance metric metric, even though there is no experimental evidence to support the idea that real programmers behave in this manner. Furthermore, the very notion of finding the true flaw is ill-defined when the bug is a sin of omission. A forgotten conditional branch, for example, corresponds to a PDG node that should have been present but is not. How can we measure the distance to a node that is not there? Better models of developer behavior are needed, and unfortunately modeling humans is well outside the domain of static program analysis.

4.2 Static Analysis Potential

In each of the examples given above, static analyses were not being applied in isolation. Rather, they were used in conjunction with statistical models built from dynamic data. Static and dynamic/statistical approaches have complementary strengths and weaknesses. A static analysis may provide strong guarantees (modulo loose C semantics) for a limited set of questions, while dynamic/statistical information can provide best-estimate guesses for nearly any question, but never with absolute certainty. If we combine the two carefully, we may achieve the best of both worlds.

Dynamic data can be used as a hypothesis generator to drive deep static analyses. Nimmer and Ernst's fusion of Daikon and ESC/Java is a classic instance of this style of dynamic-to-static feedback [26]. In the CBI context, bug predictors identified in dynamic data could suggest initial conditions on program state that warrant closer static inspection. If failures are common when `p` is null on line 94, let static analysis explore the antecedent causes or subsequent implications of that condition. Conditioned slicing, for example, may be appropriate for pursuing such leads [27], if it can be made to work for C programs of realistic size and complexity.

Useful information can also flow from the static world to the dynamic/statistical world. We suggested earlier that static analyses could prove some instrumentation redundant, and therefore removable. A related idea would be to use static analysis to reconstruct some of the data omitted by sparse sampling. To take one very simple example, any observation at a given instrumentation site reveals that the control-flow dominators of that site must also have been executed, even if sparse sampling caused this fact to be omitted from the raw feedback data. Deeper static analysis could infer missing information about data values as well as control flow. Recovering missing data effectively increases the sampling rate, resulting in a less noisy dataset for statistical

Table 1. Applications in CBI's public deployment, illustrating the infeasibility of whole-program analysis. The count of plug-ins provided includes only those that are part of the main application distribution. Any number of additional plug-ins could be provided by third parties.

Application	Lines of Code	Shared Libraries Used	Plug-Ins Provided
Evolution	441,644	107	45
GIMP	854,530	50	188
GNOME Panel	69,164	82	0
Gnumeric	351,461	85	36
Nautilus	137,394	89	0
Pidgin	387,962	56	56
Rhythmbox	133,281	95	12
SPIM & XSPIM	28,139	4 & 18	(not extensible)

modeling. Several key questions remain unanswered about this idea. It is not clear that available model checkers and theorem provers can scale up to the problem sizes that arise from this sort of analysis. (Indeed, our own preliminary exploration suggests that they do not.) Additionally, missing-data reconstruction introduces bias, as not all missing data is equally easy to infer from available evidence. Whether this bias fouls the results of statistical models, and how it can be compensated for, remain unknown.

Lastly and most speculatively, perhaps richer statistical models could draw simultaneously from both static and dynamic sources of information, instead of merely feeding one into the other. *Statistical relational learning* describes a broad class of methods for building statistical models over domains with rich internal structure [28]. Programs have rich internal structure, and research on static program analysis offers myriad strategies for extracting that structure. Exposing that static structure in a way that allows principled integration with dynamic feedback data may allow tremendous advances in program understanding and debugging. I will be the first to admit that I do not know how to do this . . . yet. But I intend to find out.

5 A Closing Rant on Analysis Robustness

Invited papers should stir things up a bit. In case this paper has not already done so, I will now indulge in a closing rant likely to agitate (if not offend) many readers.

One practical difficulty in using static analyses with CBI is that many implementations of interesting static analyses don't actually work. They worked at one time, on a few small examples sufficient to write a paper. But give them tens of thousands of lines of real C code and many analysis implementations simply fall apart. The more interesting the analysis in theory, the more brittle its implementation tends to be in practice.

One common and brittle assumption is that whole-program analysis works for mainstream applications. In my subjective experience, it does not. Even if sheer code size were not a problem, I do not have a single real application of interest where all code is available at compilation/analysis time. Table 1 summarizes applications now in CBI's public deployment. Observe that every application uses numerous shared libraries, and that all but one (SPIM/XSPIM) can be extended at run time through plug-ins. Thus, the idea that whole-program analysis can see all application code is simply a myth.

Even among the code that is present for analysis, real world software is not always pretty. I am no C apologist, and I look forward to C's eventual replacement by stricter languages that are more amenable to analysis. Until that happens, I need analyses that handle the full, horrific glory that is C: pointer casting, threads, stack-unwinding `longjmp`, dynamic code loading . . . the whole terrifying bag of C tricks. This is the language as it is used in the real world, and this is the language that a static analysis must handle if it is to be used with CBI in the near term.

6 Conclusion

Cooperative Bug Isolation operates in an messy world of unsafe languages, corrupted heaps, non-deterministic failures, and incomplete data. Faced with such obstacles, I rarely achieve or even seek software perfection. Rather, I describe my research as trying to make software suck less. In this ugly domain, even the findings of a "sound" static analysis may not be entirely trustworthy. Historically, CBI has shied away from deep static analysis in favor of brute-force data collection and statistical modeling. However, static analyses can play an important role if applied wisely. Analysis can make instrumentation more selective and efficient before deployment, and can augment statistical modeling in numerous ways after feedback data arrives. I believe that the most powerful approaches will carefully combine static, dynamic, and statistical methods to leverage the unique strengths of each. If we can do that, then perhaps even software perfection is not to much to hope for.

References

1. Liblit, B.:Cooperative Bug Isolation (Winning Thesis of the 2005 ACM Doctoral Dissertation Competition). LNCS, vol. 4440. Springer, Heidelberg (2007)
2. Driscoll, E., Cooksey, G.: CBI++. CS706 class project, University of Wisconsin–Madison (December 2006)
3. Hunter, J., Kolpin, G., Saeed, U.: CBI instrumentation for Java bytecode. CS706 class project, University of Wisconsin–Madison (December 2005)
4. Kolpin, G.: Jikes CBI implementation details. Independent study project, University of Wisconsin–Madison (May 2006)
5. Liblit, B.: Guide to the bug isolation sampler (January 2008), `http://www.cs.wisc.edu/cbi/developers/guide/`
6. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) CC 2002 and ETAPS 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
7. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69(1-3), 35–45 (2007)
8. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: PLDI, pp. 141–154. ACM, New York (2003)
9. Arnold, M., Ryder, B.G.: A framework for reducing the cost of instrumented code. In: PLDI, pp. 168–179 (2001)

10. Hirzel, M., Chilimbi, T.: Bursty tracing: A framework for low-overhead temporal profiling (November 24, 2001)
11. Zheng, A.X., Jordan, M.I., Liblit, B., Aiken, A.: Statistical debugging of sampled programs. In: Thrun, S., Saul, L.K., Schölkopf, B. (eds.) NIPS, MIT Press, Cambridge (2003)
12. Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P.: SOBER: statistical model-based bug localization. In: Wermelinger, M., Gall, H. (eds.) ESEC/SIGSOFT FSE, pp. 286–295. ACM, New York (2005)
13. Jones, J.A., Harrold, M.J.: Empirical evaluation of the Tarantula automatic fault-localization technique. In: Redmiles, D.F., Ellman, T., Zisman, A. (eds.) ASE, pp. 273–282. ACM, New York (2005)
14. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., Jordan, M.I.: Scalable statistical bug isolation. In: Sarkar, V., Hall, M.W. (eds.) PLDI, pp. 15–26. ACM, New York (2005)
15. Zheng, A.X., Jordan, M.I., Liblit, B., Naik, M., Aiken, A.: Statistical debugging: simultaneous identification of multiple bugs. In: Cohen, W.W., Moore, A. (eds.) ICML. ACM International Conference Proceeding Series, vol. 148, pp. 1105–1112. ACM, New York (2006)
16. Wassel, H.M.G.H.: An enhanced bi-clustering algorithm for automatic multiple software bug isolation. Master's thesis, Alexandria University, Egypt (September 2007)
17. Arumuga Nainar, P., Chen, T., Rosin, J., Liblit, B.: Statistical debugging using compound Boolean predicates. In: Rosenblum, D.S., Elbaum, S.G. (eds.) ISSTA, pp. 5–15. ACM, New York (2007)
18. Jiang, L., Su, Z.: Context-aware statistical debugging: from bug predictors to faulty control flow paths. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (eds.) ASE, pp. 184–193. ACM, New York (2007)
19. Andrzejewski, D., Mulhern, A., Liblit, B., Zhu, X.: Statistical debugging using latent topic models. In: Kok, J.N., Koronacki, J., de Mántaras, R.L., Matwin, S., Mladenic, D., Skowron, A. (eds.) ECML 2007. LNCS (LNAI), vol. 4701, pp. 6–17. Springer, Heidelberg (2007)
20. Liblit, B., Aiken, A.: Building a better backtrace: Techniques for postmortem program analysis. Technical Report CSD-02-1203, University of California, Berkeley (October 2002)
21. Lal, A., Lim, J., Polishchuk, M., Liblit, B.: Path optimization in programs and its application to debugging. In: Sestoft, P. (ed.) ESOP 2006 and ETAPS 2006. LNCS, vol. 3924, pp. 246–263. Springer, Heidelberg (2006)
22. Reps, T.W., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Sci. Comput. Program. 58(1-2), 206–263 (2005)
23. Lal, A., Reps, T.W., Balakrishnan, G.: Extended weighted pushdown systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 434–448. Springer, Heidelberg (2005)
24. Cleve, H., Zeller, A.: Locating causes of program failures. In: Roman, G.C., Griswold, W.G., Nuseibeh, B. (eds.) ICSE, pp. 342–351. ACM, New York (2005)
25. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: ASE, pp. 30–39. IEEE Computer Society, Los Alamitos (2003)
26. Nimmer, J.W., Ernst, M.D.: Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. Electr. Notes Theor. Comput. Sci. 55(2) (2001)
27. Canfora, G., Cimitile, A., Lucia, A.D.: Conditioned program slicing. Information & Software Technology 40(11-12), 595–607 (1998)
28. Getoor, L., Taskar, B.: Introduction to Statistical Relational Learning (Adaptive Computation and Machine Learning). MIT Press, Cambridge (2007)

Relational Analysis of Correlation*

Jörg Bauer[1], Flemming Nielson[2], Hanne Riis Nielson[2], and Henrik Pilegaard[2]

[1] Institut für Informatik, Technische Universität München, Germany
[2] DTU Informatics, Technical University of Denmark
Kongens Lyngby, Denmark
joba@model.in.tum.de, {nielson,riis,hepi}@imm.dtu.dk

Abstract. In service-oriented computing, correlations are used to determine links between service providers and users. A correlation contains values for some variables received in a communication. Subsequent messages will only be received when they match the values of the correlation. Correlations allow for the implementation of sessions, local shared memory, gradually provided input, or input provided in arbitrary order – thus presenting a challenge to static analysis.

In this work, we present a static analysis in relational form of correlations. It is defined in terms of a fragment of the process calculus COWS that itself builds on the Fusion Calculus. The analysis is implemented and practical experiments allow us to automatically establish properties of the flow of information between services.

1 Introduction

Process calculi have proved their usefulness in describing and analysing distributed systems. It is therefore natural that they are also applied to model and analyse services made available over loosely coupled networks. *Correlation* has been identified as a useful feature to model services and hence appears in emerging service-oriented process calculi [6,7].

Correlation is an idea that stems from executable business process languages such as BPEL. In the presence of multiple concurrent instances of the same service, messages sent from a user to the provider must be delivered to the correct instance of the service. This is achieved by associating specific, already available data in the messages to maintain a unique reference to a specific service instance. For example, such data may be derived from personal information like a social security number.

Technically, in process calculi, correlation may be realised by the *decoupling of name binding and input actions* in combination with *pattern matching* in input prefixes. Decoupling was first formulated in the Fusion Calculus [15], where inputs are *not* binders. Rather, a *scope construct* $(x)P$ is the only binder binding x in P. The effect of a communication inside P may then be to *fuse* x with another name, e.g., y. The names x and y will then be considered identical. The scope of the subsequent substitution of y for x will then be all of P. This allows

* This work has been partially sponsored by the project SENSORIA, IST-2005-016004.

© Springer-Verlag Berlin Heidelberg 2008

to model local, shared state, which is essential for correlations. As argued in [4], this decoupling is impossible to encode naturally in the π-calculus.

Based on these considerations we have delineated the *Calculus for Web Services*, CWS. It is mainly a fragment of the Calculus of Orchestration of Web Services (COWS, [7]) developed in the EU project SENSORIA. In contrast to the Fusion Calculus (or D-Fusion developed in [4]), COWS is clearly service-centred and focuses on, among others, correlation. In CWS, however, we discard some of the technicalities associated with session management.

The goal of this work is not the design of a new language but the development of a relational analysis for correlation. The insights gained here should be easily transferable to any other correlation-based language for services.

Contribution. The study of process calculi is a challenging avenue for the development and application of static analysis. Such analyses often tend to fall between two extremes: rather simple 0-CFA analyses (e.g., [3]) expressing only rudimentary properties, or extremely powerful relational and polyhedral analyses (e.g., [17,5]). The latter technology seems to be mastered by only a few, and hence might not obtain widespread use, and furthermore seems to require that the calculus in question be presented in a *non-trivial normalised form* in order to aid the analysis thereby making it a major effort to apply the techniques. To be specific, the work of [17] on developing relational and polyhedral analyses for the π-calculus depends heavily on the PhD thesis of [16] that develops the non-trivial normalised form used. We therefore consider it an important achievement of our work that we populate the middle ground between the two extremes – this approach was used successfully in [9] dealing with the π-calculus. Here we apply this approach to develop a relational static analysis for correlation, which is a challenging and highly relevant aspect of process calculi for service-oriented computing. To the best of our knowledge [2], we are the first to develop a static analysis for a descendant of the Fusion Calculus.

Outline. In Section 2, we shall introduce the CWS calculus and illustrate it by giving an example of an accident service. In Section 3, we develop a static analysis in relational form for CWS. Before we report on our implementation and experimental results, we establish the correctness of our analysis expressed by two major theorems, subject reduction and adequacy. It is noteworthy that our subject reduction result does not claim that analysability is preserved under reduction, but rather that a stronger notion of "analysability of all permitted instances" is preserved under reduction. This provides a new insight in the development of static analyses of process calculi. Section 5 reports on related work and Section 6 concludes.

2 Calculus for Web Services

Table 1 defines the syntactic domains and the syntax of CWS, the Calculus for Web Services. We have three syntactic domains, *names*, *variables*, and *labels*.

Table 1. Syntax of CWS and its syntactic domains. Service invocation, request processing, as well as the variable scope are decorated with labels, $\ell \in \mathbf{Lab}$. An overline denotes tuples, for instance, $\bar{x}$ denotes tuples of variables.

$$
\begin{array}{rll}
s ::= & & \text{Services} \\
 & u \bullet u'!\bar{v}^{\ell} & \text{(invoke)} \\
| & g & \text{(input-guarded choice)} \\
| & s \mid s & \text{(parallel composition)} \\
| & (n)s & \text{(name binding box)} \\
| & [x]^{\ell} s & \text{(variable scope)} \\
| & *s & \text{(replication)} \\
 & & \\
g ::= & & \text{(input-guarded choice)} \\
 & \mathbf{0} & \text{(nil)} \\
| & p \bullet o?\bar{v}^{\ell}.s & \text{(request processing)} \\
| & g + g & \text{(choice)}
\end{array}
$$

Name	**Domain**	**Symbols**
Variables	**Var**	x, y
Names	**Name**	n, m
Partners	**Name**	p, p'
Operations	**Name**	o, o'
	Name $\cup$ **Var**	u, v
Labels	**Lab**	ℓ

While the latter merely serve as pointers into the syntax guiding the analysis specification of Section 3, names denote computational values and variables are used to bind names. In the sequel we assume that services are consistently labelled: two equally labelled actions are either both inputs – they coincide on partner, operation, and on the input length – or both outputs – they coincide on the output length.

The computational entities of CWS are called *services*. In contrast to, e.g., the π-calculus, communication endpoints in CWS are pairs $p \bullet o$ of a *partner* and an *operation* modeling several services available at the same site. Communication endpoints at reception sites are *statically determined*, because a service is supposed to know itself. In contrast, received partner and operation names may be used for subsequent service invocation. Additionally, CWS has output – rendered *asynchronous* by the missing continuation after the service invocation – input actions, input-guarded choice, parallel composition, name binding, variable scope declaration, and replication; each with the expected meaning. Names in input prefixes are used for pattern matching.

Variable binding and global scope are defined similarly to the Fusion Calculus and differently from π-like calculi: Variables are *not* bound at input actions, the only variable binder is the *scope construct* $[x]^{\ell}s$. If variable x occurs in an input action a, at which name n may be received, then the scope of the induced substitution, $[x \mapsto n]$, is the whole of s; not just the continuation of a.

2.1 An Accident Service

Our running example is given in Table 2. It describes an arbitrary number of cars (1-3) that have subscribed to an *accident service* (4-10). Each car is equipped with a GPS device tracking its position. In case an on-board sensor detects any abnormal behaviour, the alarm operation of the service centre is invoked (1) by

Table 2. An accident service

$*(\mathsf{self})\ \ *(\mathsf{gps})\ \ *(\mathsf{sid})$	$\mathsf{p}_s \bullet \mathsf{o}_{alarm}!\langle \mathsf{acc}, \mathsf{self}, \mathsf{sid}\rangle^{1}$	(1)
	$\mid\ (\mathsf{self} \bullet \mathsf{o}_{confirm}?\langle \mathsf{sid}\rangle^{2}.\mathsf{p}_s \bullet \mathsf{o}_{confirm}!\langle \mathsf{ok}, \mathsf{self}, \mathsf{sid}\rangle^{3}\ +$	(2)
	$\mathsf{self} \bullet \mathsf{o}_{confirm}?\langle \mathsf{sid}\rangle^{4}.\mathsf{p}_s \bullet \mathsf{o}_{confirm}!\langle \mathsf{ko}, \mathsf{gps}, \mathsf{sid}\rangle^{5})$	(3)
$*[x_{info}]^{6}\ \ [x_{id}]^{7}\ \ [x_{reply}]^{8}\ \ [x_{sid}]^{9}$	$\mathsf{p}_s \bullet \mathsf{o}_{alarm}?\langle \mathsf{acc}, x_{id}, x_{sid}\rangle^{10}.$	(4)
	$x_{id} \bullet \mathsf{o}_{confirm}!\langle x_{sid}\rangle^{11}$	(5)
	$\mid\ \mathsf{p}_s \bullet \mathsf{o}_{confirm}?\langle x_{reply}, x_{info}, x_{sid}\rangle^{12}.$	(6)
	$\mathsf{p}_s \bullet \mathsf{o}_{Sos}!\langle x_{reply}, x_{info}, x_{sid}\rangle^{13}$	(7)
	$\mid\ \mathsf{p}_s \bullet \mathsf{o}_{Sos}?\langle \mathsf{ko}, x_{info}, x_{sid}\rangle^{14}.$	(8)
	$\mathsf{p}_{amb} \bullet \mathsf{o}_{Sos}!\langle x_{info}, x_{sid}\rangle^{15}$	(9)
	$\mid\ \mathsf{p}_s \bullet \mathsf{o}_{Sos}?\langle \mathsf{ok}, x_{info}, x_{sid}\rangle^{16}.\mathbf{0}$	(10)

sending an accident message with the identity self of the driver and the nonce sid, which prevents malign session interference. The centre processes this request (4) and asks for confirmation (5), whereupon the alarm is either revoked (ok,2) or confirmed (ko,3) by the driver. The driver identity is attached to the revocation message, while the current GPS position is attached to the confirmation message. Then the centre processes the answer and invokes another internal service (6,7). Depending on the driver's answer either an ambulance is called and informed about the location of the accident (8,9), or a false alarm is detected (10).

Note that there may be many cars around having a number of false alarms involving different locations. Due to the safety-critical nature of this example, the service centre must be able to handle many of these service calls concurrently without mixing up information from different sessions. In particular, we would like to guarantee (and shall indeed show) that:

1. The ambulance is not called, when an ok was received.
2. Messages sent to the ambulance always contain GPS information. Note, that the validity of this property is not obvious, since x_{info} may be bound to both driver identities and positions at run-time.
3. If several cars employ the accident service at once, then only the positions of the ones confirming the accident are reported to the ambulance.

2.2 Labelled Transition System for CWS

To define the semantics of CWS we employ a notion of structural congruence defined as the least congruence that incorporates the axioms of Table 3, contains disciplined α-renaming of names[1], and asserts that choice and parallel are associative, commutative, and have $\mathbf{0}$ as neutral element. We do *not* allow the extrusion of *variable scopes*. Rather, [Binder$_4$] allows the binding boxes of *names* to migrate freely in and out of variable scopes. *Substitutions*, σ, are mappings

[1] For disciplined α-renaming, we identify each name with its defining syntactic occurrence, its canonical name, thus partitioning the name space into finitely many equivalence classes rendering canonical names stable under evaluation.

Table 3. An excerpt of the CWS structural congruence rules. For a given service expression s, $fn(s)$ denotes the set of free names.

$*\mathbf{0} \equiv \mathbf{0}$	[Repl$_1$]
$*s \equiv s \mid *s$	[Repl$_2$]
$(n)\mathbf{0} \equiv \mathbf{0}$	[Binder$_1$]
$(n)(m)s \equiv (m)(n)s$	[Binder$_2$]
$s_1 \mid (n)s_2 \equiv (n)(s_1 \mid s_2)$ if $n \notin fn(s_1)$	[Binder$_3$]
$[x](n)s \equiv (n)[x]s$	[Binder$_4$]

with domain **Var** $\rightarrow$ **Name**. The application of a substitution, $[x \mapsto n]$, to a service s is written $s \cdot [x \mapsto n]$ and replaces free occurrences of x in s by n. The disjoint union of two substitutions σ_1 and σ_2 is written $\sigma_1 \uplus \sigma_2$ and the substitution with empty domain is written $\emptyset$. A tuple $\bar{u}$ of names and variables is matched against a tuple $\bar{n}$ of names as follows (yielding a substitution):

$$\mathcal{M}(x,n) = x \mapsto n \qquad \mathcal{M}(n,n) = \emptyset \qquad \frac{\mathcal{M}(u_1,n_1) = \sigma_1 \quad \mathcal{M}(\bar{u}_2,\bar{n}_2) = \sigma_2}{\mathcal{M}((u_1,\bar{u}_2),(n_1,\bar{n}_2)) = \sigma_1 \uplus \sigma_2}$$

The semantics of CWS (Table 4) is given as a labelled transition system using transition labels α:

$$\alpha ::= (p \bullet o) \lhd \bar{n}^{\ell} \ \mid \ (p \bullet o) \rhd \bar{u}^{\ell} \ \mid \ p \bullet o\lfloor \sigma \rfloor \bar{u}^{\ell_i} \bar{n}^{\ell_o}$$

Transition labels $(p \bullet o) \lhd \bar{n}^{\ell}$ and $(p \bullet o) \rhd \bar{u}^{\ell}$ result from applying rules for invocation and reception of names, respectively, that is, rules [Inv] and [Rec]. They can engage in a communication ([Com]) giving rise to a transition label $p \bullet o\lfloor \sigma \rfloor \bar{u}^{\ell_i} \bar{n}^{\ell_o}$, where σ is the resulting substitution. Note that the only transition labels being *observable* are those of the form $p \bullet o\lfloor \emptyset \rfloor \bar{u}^{\ell_i} \bar{n}^{\ell_o}$. We use $d(\alpha)$ to denote the set of names and variables occurring in α; if $\alpha = p \bullet o\lfloor \sigma \rfloor \bar{u}^{\ell_i} \bar{n}^{\ell_o}$ then $d(\alpha)$ contains only the names and variables in the domain and codomain of σ.

At top level, the semantics is only defined for *closed services*, that is, no free variables may occur when a computation step is applied. Deeper in the inference tree there may of course be free occurrences of variables. A substitution $[x \mapsto n]$ is only applied, when the enclosing scope of x is met ([Del$_{sub}$]). This ensures that the effect of a communication is visible *globally* within a scope and allows to model shared memory of a session instance. Services can proceed normally under name bindings or variable scopes, unless the enclosing entity is mentioned by the transition label ([Name] and [Scope]). Note that for an output to occur all variables in it must have been substituted by names earlier ([Inv]).

Comparison with COWS. CWS lacks the orchestration constructs (kill and protect) found in COWS, because we focus on correlation rather than on orchestration. The semantics of CWS and COWS are comparable up to rule [Com]: Let two different communications both yield a valid match. While our rule picks non-deterministically from the set of choices, the COWS semantics [7] imposes a constraint, *noc*, selecting the match that gives rise to the smallest substitution.

Table 4. CWS operational semantics

$$[\textsc{Inv}]\quad p \bullet o!\bar{n}^{\ell} \xrightarrow{(p\bullet o)\lhd\bar{n}^{\ell}} \mathbf{0} \qquad\qquad [\textsc{Rec}]\quad p \bullet o?\bar{u}^{\ell}.s \xrightarrow{(p\bullet o)\rhd\bar{u}^{\ell}} s$$

$$[\textsc{Choice}]\quad \frac{g_1 \xrightarrow{\alpha} s}{g_1 + g_2 \xrightarrow{\alpha} s} \qquad\qquad [\textsc{Del}_{sub}]\quad \frac{s \xrightarrow{p\bullet o\lfloor\sigma\uplus[x\mapsto n]\rfloor\,\bar{u}^{\ell_i}\,\bar{n}^{\ell_o}} s'}{[x]^{\ell}s \xrightarrow{p\bullet o\lfloor\sigma\rfloor\,\bar{u}^{\ell_i}\,\bar{n}^{\ell_o}} s'\cdot[x\mapsto n]}$$

$$[\textsc{Name}]\quad \frac{s \xrightarrow{\alpha} s' \qquad n \notin d(\alpha)}{(n)s \xrightarrow{\alpha} (n)s'} \qquad\qquad [\textsc{Scope}]\quad \frac{s \xrightarrow{\alpha} s' \qquad x \notin d(\alpha)}{[x]s \xrightarrow{\alpha} [x]s'}$$

$$[\textsc{Com}]\quad \frac{s_1 \xrightarrow{(p\bullet o)\rhd\bar{u}^{\ell_i}} s_1' \qquad s_2 \xrightarrow{(p\bullet o)\lhd\bar{n}^{\ell_o}} s_2' \qquad \mathcal{M}(\bar{u},\bar{n}) = \sigma}{s_1 \mid s_2 \xrightarrow{p\bullet o\lfloor\sigma\rfloor\,\bar{u}^{\ell_i}\,\bar{n}^{\ell_o}} s_1' \mid s_2'}$$

$$[\textsc{Par}]\quad \frac{s_1 \xrightarrow{\alpha} s_1'}{s_1 \mid s_2 \xrightarrow{\alpha} s_1' \mid s_2} \qquad\qquad [\textsc{Cong}]\quad \frac{s \equiv s_1 \qquad s_1 \xrightarrow{\alpha} s_2 \qquad s_2 \equiv s'}{s \xrightarrow{\alpha} s'}$$

In case of equally long substitutions, the choice is random. In [7], this feature is used to distinguish (less instantiated) service definitions from (more instantiated) service instances. However, we omit this feature from our semantics, because it is not analysed, and our analysis would still produce a valid over-approximation if we used it. In order to prevent malign session interferences we adapt what is considered good style in communication protocols, where one relies on the use of nonces or shared secrets (or indeed shared keys) to ensure that sessions do not interfere. Finally, note that it is indeed possible to implement the Fusion-like scoping by using explicit environments in the semantics – however, as anywhere else, we decided to follow COWS as closely as possible.

3 A Relational Analysis for CWS

In this section, we develop a relational static analysis [10] for CWS: For each program label ℓ – invocation, processing, and variable scope – we compute sets of tuples of names to which the variables that are in scope at ℓ may be bound at run-time. Tracking *sets* of tuples makes the analysis relational, while an independent analysis would track tuples of sets and loose track of which variables are bound at the same time.

Auxiliary Information. A *label environment*, L, is defined as a mapping

$$\mathsf{L} : \mathbf{Lab} \to (\mathbf{Var} \times \mathbf{Lab})^*$$

Table 5. Label environment $\mathcal{L}_{\bar{x}}[\![s]\!]$ and flow information $\mathcal{F}[\![s]\!]$

$$\begin{aligned}
\mathcal{L}_{\bar{\chi}}[\![\mathbf{0}]\!] &= [\,] \\
\mathcal{L}_{\bar{\chi}}[\![u \bullet u'!\bar{v}^{\ell}]\!] &= [\ell \mapsto \bar{\chi}] \\
\mathcal{L}_{\bar{\chi}}[\![p \bullet o?\bar{u}^{\ell}.s]\!] &= \mathcal{L}_{\bar{\chi}}[\![s]\!] \uplus [\ell \mapsto \bar{\chi}] \\
\mathcal{L}_{\bar{\chi}}[\![(n)s]\!] &= \mathcal{L}_{\bar{\chi}}[\![s]\!] \\
\mathcal{L}_{\bar{\chi}}[\![*s]\!] &= \mathcal{L}_{\bar{\chi}}[\![s]\!] \\
\mathcal{L}_{\bar{\chi}}[\![[x]^{\ell} s]\!] &= \mathcal{L}_{\bar{\chi}x^{\ell}}[\![s]\!] \uplus [\ell \mapsto \bar{\chi}] \\
\mathcal{L}_{\bar{\chi}}[\![s_1 \mid s_2]\!] &= \mathcal{L}_{\bar{\chi}}[\![s_1]\!] \uplus \mathcal{L}_{\bar{\chi}}[\![s_2]\!] \\
\mathcal{L}_{\bar{\chi}}[\![s_1 + s_2]\!] &= \mathcal{L}_{\bar{\chi}}[\![s_1]\!] \uplus \mathcal{L}_{\bar{\chi}}[\![s_2]\!]
\end{aligned}$$

$$\begin{aligned}
\mathcal{F}[\![\mathbf{0}]\!] &= ([\,], \emptyset) \\
\mathcal{F}[\![u \bullet u'!\bar{v}^{\ell}]\!] &= ([\,], \{\ell\}) \\
\mathcal{F}[\![p \bullet o?\bar{u}^{\ell}.s]\!] &= \text{let } (F, E) = \mathcal{F}[\![s]\!] \\
&\quad\ \text{in } \ (F \uplus [\ell \mapsto E], \{\ell\}) \\
\mathcal{F}[\![(n)s]\!] &= \mathcal{F}[\![s]\!] \\
\mathcal{F}[\![*s]\!] &= \mathcal{F}[\![s]\!] \\
\mathcal{F}[\![[x]^{\ell} s]\!] &= \text{let } (F, E) = \mathcal{F}[\![s]\!] \\
&\quad\ \text{in } \ (F \uplus [\ell \mapsto E], \{\ell\}) \\
\mathcal{F}[\![s_1 \mid s_2]\!] &= \mathcal{F}[\![s_1 + s_2]\!] \\
&= \text{let } (F_1, E_1) = \mathcal{F}[\![s_1]\!] \\
&\qquad\quad (F_2, E_2) = \mathcal{F}[\![s_2]\!] \\
&\quad\ \text{in } \ (F_1 \uplus F_2, E_1 \cup E_2)
\end{aligned}$$

that to each label ℓ associates a sequence $\bar{\chi} \in (\mathbf{Var} \times \mathbf{Lab})^*$ of pairs of variables and labels that have been introduced *before* this point in the process; the label indicates the exact scope where the variable was introduced. Formally, we shall take $\mathsf{L} = \mathcal{L}_{\epsilon}[\![s]\!]$, where $\mathcal{L}_{\bar{\chi}}$ is defined in Table 5. Here we write $\uplus$ for joining two mappings with *disjoint* domains and $[\,]$ for the mapping with empty domain. The notation $\bar{\chi}_1\bar{\chi}_2$ stands for the concatenation of $\bar{\chi}_1$ and $\bar{\chi}_2$. Furthermore, we write $x\#\mathsf{L}.\ell$ for the label, at which x at position ℓ was defined, taking into account that the most recently introduced definition of x is the rightmost:

$$x\#\langle x_1^{\ell_1}, \ldots, x_k^{\ell_k}\rangle = \ell_t, \text{ if } t = \max\{i \mid x_i = x\}$$

This notion is only defined, when x is among the x_i, which will always be the case, when we use it.

Flow of control in a service s is represented a *flow mapping* F that to each label ℓ associates the set of labels that will become visible once the action labelled ℓ has been executed; thus

$$\mathsf{F} : \mathbf{Lab} \hookrightarrow \mathcal{P}(\mathbf{Lab})$$

Function $\mathcal{F}$ in Table 5 computes the flow mapping, F, together with the set E of *visible labels* of a service s, where $(\mathsf{F}, \mathsf{E}) = \mathcal{F}[\![s]\!]$. When applying F and L to labels, we shall write $\mathsf{L}.\ell$ and $\mathsf{F}.\ell$ instead of $\mathsf{L}(\ell)$ and $\mathsf{F}(\ell)$.

Example 1. Label environment computed from the running example in Table 2:

$$\begin{gathered}
\mathsf{L}.1 = \mathsf{L}.2 = \mathsf{L}.3 = \mathsf{L}.4 = \mathsf{L}.5 = \mathsf{L}.6 = \epsilon \qquad\qquad \mathsf{L}.7 = \langle(x_{info}, 6)\rangle \\
\mathsf{L}.8 = \langle(x_{info}, 6), (x_{id}, 7)\rangle \qquad \mathsf{L}.9 = \langle(x_{info}, 6), (x_{id}, 7), (x_{reply}, 8)\rangle \\
\mathsf{L}.10 = \mathsf{L}.11 = \cdots = \mathsf{L}.16 = \langle(x_{info}, 6), (x_{id}, 7), (x_{reply}, 8), (x_{sid}, 9)\rangle
\end{gathered}$$

In the sequel, we shall often write pairs like $(x_{info}, 6)$ using superscript as in x_{info}^6. Two examples of flow information are $\mathsf{F}.9 = \{10, 12, 14, 16\}$ and $\mathsf{F}.10 = \{11\}$. Finally, the set of visible labels for the running example is $\{1, 2, 4, 6\}$. $\square$

Analysis Domain. The *abstract environments* $\hat{\mathsf{R}}$ of the analysis will, given a label ℓ, return a set of sequences. Each element of such a sequence can either be a name or the *undefined* symbol $\perp$. The latter denotes cases, where a variable has not been assigned a value yet. This helps to track gradually provided inputs. The length of these sequences will equal that of $\mathsf{L}.\ell$. We will thus determine the potential values of the variables at the point determined by ℓ and take:

$$\hat{\mathsf{R}} : \mathbf{Lab} \to \mathcal{P}(\mathbf{Name}_{\perp}^{*})$$

where $\mathbf{Name}_{\perp} = \mathbf{Name} \cup \{\perp\}$.[2] We shall use w to denote elements of $\mathbf{Name}_{\perp}$. If $\hat{\mathsf{R}}.\ell = \emptyset$ it means that the program point ℓ is not reachable; if $\hat{\mathsf{R}}.\ell = \{\epsilon\}$ then also $\mathsf{L}.\ell = \epsilon$ and it means that no variable has been introduced at that program point.

The potential values of a variable x at the label ℓ are computed by $\Pi_{x@\mathsf{L}.\ell}(\bar{w})$, where $\bar{w}$ has the same length as $\mathsf{L}.\ell$ and x occurs in $\mathsf{L}.\ell$. In cases with more than one occurrence of x in $\mathsf{L}.\ell$ we select the rightmost, i.e., the most recently declared one. Formally:

$$\Pi_{x@\mathsf{L}.\ell}(\bar{w}) = w_t$$

where $\bar{w} = \langle w_1, \dots, w_k \rangle$, $\mathsf{L}.\ell = \langle x_1^{\ell_1}, \dots, x_k^{\ell_k} \rangle$, and $t = \max\{i \mid x_i = x\}$. Note, that the result may be $\perp$. The operation is trivially extended to names, n: $\Pi_{n@\mathsf{L}.\ell}(\bar{w}) = n$ Also, it is extended to sequences, $\bar{u}$, of variables and names, as in $\Pi_{\bar{u}@\mathsf{L}.\ell}(\bar{w})$, and to sets, R, of such sequences, i.e., $\Pi_{\bar{u}@\mathsf{L}.\ell}(R)$. The abstract communication cache

$$\hat{\mathsf{K}} \subseteq \mathbf{Name} \times \mathbf{Name} \times \mathbf{Name}^{*}$$

records the tuples of names that potentially are communicated over the channels. Elements of the domain are triples representing a partner, an operation, and the tuple of names communicated. In $\hat{\mathsf{K}}$, we keep track of names only, there is no $\perp$ involved.

Analysis Specification. The *judgements* of the analysis have the form

$$\hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} s$$

where L, F, $\hat{\mathsf{R}}$, and $\hat{\mathsf{K}}$ are as above. Intuitively, the judgements defined in Table 6 determine whether a given pair $(\hat{\mathsf{R}}, \hat{\mathsf{K}})$ is a valid analysis result. The computation of such a pair is described in Section 4.2.

The first five rules are recursive cases. The rule [RInv] deals with invocations. It looks up the available bindings of the involved variables in $\hat{\mathsf{R}}$ and records the resulting tuples in the abstract communication cache, $\hat{\mathsf{K}}$. Since the lookup may yield undefined values, we need to intersect with $\mathbf{Name}^{*}$.

Rule [RScope] extends all variable bindings available at scope definition ℓ with a single $\perp$ and passes the information on to the program points following ℓ. This reflects that a newly introduced variable is not yet bound.

[2] Again, all the names tracked in the analysis are canonical.

Table 6. Specification of the analysis judgement $\hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} s$

$$[\textsc{RNil}]\quad \hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} \mathbf{0} \qquad [\textsc{RRep}]\quad \frac{\hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} s}{\hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} *s} \qquad [\textsc{RName}]\quad \frac{\hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} s}{\hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} (n)s}$$

$$[\textsc{RPar}]\quad \frac{\hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} s_1 \quad \hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} s_2}{\hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} s_1 \mid s_2} \qquad [\textsc{RChoice}]\quad \frac{\hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} s_1 \quad \hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} s_2}{\hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} s_1 + s_2}$$

$$[\textsc{RInv}]\ \hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} u \bullet u'!\bar{v}^{\ell} \quad \textbf{if } \Pi_{uu'\bar{v}@\mathsf{L}.\ell}(\hat{\mathsf{R}}.\ell) \cap \mathbf{Name}^* \subseteq \hat{\mathsf{K}}$$

$$[\textsc{RScope}]\ \frac{\hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} s}{\hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} [x]^{\ell} s} \quad \textbf{if } \forall \ell' \in \mathsf{F}.\ell : \hat{\mathsf{R}}.\ell \times \{\bot\} \subseteq \hat{\mathsf{R}}.\ell'$$

$$[\textsc{RRec}]\ \frac{X \neq \emptyset \ \Rightarrow\ \hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} s}{\hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} p \bullet o?\bar{u}^{\ell}.s} \quad \textbf{if} \begin{array}{l} \forall \ell' \in \mathsf{F}.\ell : X \subseteq \hat{\mathsf{R}}.\ell' \\ \forall x \in \{\bar{u}\}\ \forall \ell_x \in (\mathsf{F}.(x\#\mathsf{L}.\ell)) : Y_x \subseteq \hat{\mathsf{R}}.\ell_x \end{array}$$

$$\begin{array}{ll} \textbf{where} & X = \{\bar{w} \in \hat{\mathsf{R}}.\ell \mid \Pi_{po\bar{u}@\mathsf{L}.\ell}(\bar{w}) \in \hat{\mathsf{K}}\} \\ \textbf{and} & Y_x = \{\bar{w}n \in \mathbf{Name}_{\bot}^* \mid \bar{w} \in \hat{\mathsf{R}}.(x\#\mathsf{L}.\ell) \wedge \\ & \qquad\qquad \exists \bar{w}' \in \mathbf{Name}_{\bot}^* : \Pi_{po\bar{u}@\mathsf{L}.\ell}(\bar{w}n\bar{w}') \in \hat{\mathsf{K}}\ \} \end{array}$$

Finally, the rule [RRec] describes two different flows. First, it ensures that all possible bindings at ℓ that may lead to a communicated tuple will indeed flow to the subsequent visible labels (expressed in the set X); only if this set is non-empty is it possible to perform the input, hence the test $X \neq \emptyset$ is a reachability condition for the continuation s. Second, for each variable x of the input pattern, we record in the set Y_x which names may be bound to x by any communication that matches this input pattern. This information flows to the labels just after the scope at which x was introduced (denoted by $\mathsf{F}.(x\#\mathsf{L}.\ell)$).

In addition to the satisfaction of the analysis judgement, we require some initialisation information, stating that ϵ is available at all globally visible labels:

Definition 1. *Let s be a service, $(\mathsf{F}, \mathsf{E}) = \mathcal{F}[\![s]\!]$ its flow information, and $\mathsf{L} = \mathcal{L}_{\epsilon}[\![s]\!]$ its label environment. A pair $(\hat{\mathsf{R}}, \hat{\mathsf{K}})$ is an* acceptable analysis estimate *for s, if and only if $\hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} s$ and $\epsilon \in \hat{\mathsf{R}}.\ell$ for all $\ell \in \mathsf{E}$.*

Example 2. An acceptable analysis estimate – and in fact the least, that is, most precise one – of the running example of Table 2 comprises:

$$\begin{array}{rl} \hat{\mathsf{K}} = \{ & \langle \mathsf{p}_s, \mathsf{o}_{alarm}, \mathsf{acc}, \mathsf{self}, \mathsf{sid}\rangle, \langle \mathsf{p}_s, \mathsf{o}_{confirm}, \mathsf{ok}, \mathsf{self}, \mathsf{sid}\rangle, \\ & \langle \mathsf{p}_s, \mathsf{o}_{confirm}, \mathsf{ko}, \mathsf{gps}, \mathsf{sid}\rangle, \langle \mathsf{self}, \mathsf{o}_{confirm}, \mathsf{sid}\rangle, \langle \mathsf{p}_s, \mathsf{o}_{Sos}, \mathsf{ok}, \mathsf{self}, \mathsf{sid}\rangle, \\ & \langle \mathsf{p}_s, \mathsf{o}_{Sos}, \mathsf{ko}, \mathsf{gps}, \mathsf{sid}\rangle, \langle \mathsf{p}_{amb}, \mathsf{o}_{Sos}, \mathsf{gps}, \mathsf{sid}\rangle\ \} \end{array}$$

Regarding the abstract environments, we state only $\hat{\mathsf{R}}.13$ and $\hat{\mathsf{R}}.15$ explicitly, because they are most relevant with respect to the properties we are interested in. Recall that both $\mathsf{L}.13$ and $\mathsf{L}.15$ amount to $\langle (x_{info}, 6), (x_{id}, 7), (x_{reply}, 8), (x_{sid}, 9)\rangle$.

$$\begin{array}{l} \hat{\mathsf{R}}.13 = \{\langle \mathsf{self}, \mathsf{self}, \mathsf{ok}, \mathsf{sid}\rangle, \langle \mathsf{self}, \bot, \mathsf{ok}, \mathsf{sid}\rangle, \langle \mathsf{gps}, \mathsf{self}, \mathsf{ko}, \mathsf{sid}\rangle, \langle \mathsf{gps}, \bot, \mathsf{ko}, \mathsf{sid}\rangle\} \\ \hat{\mathsf{R}}.15 = \{\langle \mathsf{gps}, \mathsf{self}, \mathsf{ko}, \mathsf{sid}\rangle, \langle \mathsf{gps}, \mathsf{self}, \bot, \mathsf{sid}\rangle, \langle \mathsf{gps}, \bot, \mathsf{ko}, \mathsf{sid}\rangle, \langle \mathsf{gps}, \bot, \bot, \mathsf{sid}\rangle\} \end{array}$$

Reconsider the three properties stated in Section 2.1. Property 1 states that the ambulance is not called, when an ok was received. The ambulance is called at label 15. In the analysis result, we can see that the value of x_{reply} (the third position in the tuples) cannot be ok proving property 1. Note that at label 13, ok may still occur.

Property 2 requires that an ambulance is only called with location information. It is shown by inspecting $\hat{\mathsf{K}}$, which over-approximates all messages sent: All message involving p_{amb} in $\hat{\mathsf{K}}$ contain gps only.

Property 3 is not so easy to show for the analysis as is. We are able to establish it, if we unfold the definition of cars a finite number of times, though. The analysis will then show, that malign interferences are prevented by the session id sid. The formal result justifying our reasoning about properties in this example is stated in Theorem 2 below. □

4 Properties of the Analysis

In this section, we establish the formal correctness of our analysis. We start by defining a *correctness predicate*, which states the analysability of *all permitted substitutions* of an analysable service, and show that its validity is preserved under observable computation steps. This constitutes our subject reduction result (Theorem 1). It is shown in [1] that mere analysability is not preserved under reduction. Theorem 2 states that all actually sent messages and all potential variable bindings are correctly recorded in the analysis. We conclude this section by reporting on experiments using the implementation of our analysis. The feasibility of this implementation relies on the Moore family property of the set of all acceptable analyses guaranteeing the existence of least (most precise) solutions (Theorem 3).

4.1 Correctness

In the following we assume an arbitrary but fixed given program $s_\star$, as well as its flow information $\mathsf{F} = \mathcal{F}[\![s_\star]\!]$, its label and its visible labels $(\mathsf{L}, \mathsf{E}) = \mathcal{L}_\epsilon[\![s_\star]\!]$. Moreover, we define the notion $\mathcal{E}(s)$ of the *exposed actions* of a service s, which is a set of input and output prefixes, such that $\mathcal{E}(u \bullet u'!\bar{v}^\ell) = \{u \bullet u'!\bar{v}^\ell\}$, $\mathcal{E}(p \bullet o?\bar{v}^\ell.s) = \{p \bullet o?\bar{v}^\ell.s\}$, $\mathcal{E}(*s) = \mathcal{E}((n)s) = \mathcal{E}([x]s) = \mathcal{E}(s)$, $\mathcal{E}(\mathbf{0}) = \emptyset$, $\mathcal{E}(s_1 \mid s_2) = \mathcal{E}(s_1 + s_2) = \mathcal{E}(s_1) \cup \mathcal{E}(s_2)$. Finally, we define an extended version of substitution. Let $s = p \bullet o?\bar{v}^\ell.s'$ or $s = u \bullet u'!\bar{u}^\ell$ and let $\bar{w} \in \hat{\mathsf{R}}.\ell$ where $\mathsf{L}.\ell = \langle x_1, \ldots, x_k\rangle$ and $\bar{w} = \langle w_1, \ldots, w_k\rangle$. Then we define

$$s[\bar{w}/\mathsf{L}.\ell] = (\ldots(s \cdot [x_k \mapsto w_k]) \cdot \ldots) \cdot [x_1 \mapsto w_1]$$

where $s \cdot [x \mapsto \bot] = s$.

Intuitively, the correctness predicate of Definition 2 holds of a service s obtained by semantic reduction from $s_\star$, if all exposed actions s' of s have a counterpart s'' in $s_\star$, such that s'' is correctly analysed and such that s'' is congruent to s' when instantiating it with information computed by the analysis. In other words, the correctness predicate describes the analysability of permitted substitution instances.

Definition 2 (Correctness Predicate). *A service s satisfies the correctness predicate with respect to $s_\star$, written $\hat{\mathsf{R}}, \hat{\mathsf{K}} \models^{s_\star} s$ if and only if (1-4) hold.*

1. $\hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} s_\star$
2. $\forall \ell \in \mathsf{E} : \epsilon \in \hat{\mathsf{R}}.\ell$
3. *For all $p \bullet o?\bar{u}^\ell.s' \in \mathcal{E}(s)$ there exists a subexpression $p \bullet o?\bar{v}^\ell.s''$ of $s_\star$ s.t.*
   - $\hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} p \bullet o?\bar{v}^\ell.s''$ *and*
   - $\exists \bar{w} \in \hat{\mathsf{R}}.\ell : p \bullet o?\bar{u}^\ell.s' \equiv p \bullet o?\bar{v}^\ell.s''[\bar{w}/\mathsf{L}.\ell]$
4. *For all $v \bullet v'!\bar{v}^\ell \in \mathcal{E}(s)$ there exists a subexpression $u \bullet u'!\bar{u}^\ell$ of $s_\star$ s.t.*
   - $\hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} u \bullet u'!\bar{u}^\ell$ *and*
   - $\exists \bar{w} \in \hat{\mathsf{R}}.\ell : v \bullet v'!\bar{v}^\ell \equiv u \bullet u'!\bar{u}^\ell[\bar{w}/\mathsf{L}.\ell]$

Lemma 1 states some obvious compositionality properties of the correctness predicate. The proof is straightforward from Definition 2.

Lemma 1 (Compositionality). *Let s, s_1, s_2 be services, x a variable and n a name. It holds:*

- $\hat{\mathsf{R}}, \hat{\mathsf{K}} \models^{s_\star} s_1 \mid s_2$ *if and only if* $\hat{\mathsf{R}}, \hat{\mathsf{K}} \models^{s_\star} s_1$ *and* $\hat{\mathsf{R}}, \hat{\mathsf{K}} \models^{s_\star} s_2$.
- $\hat{\mathsf{R}}, \hat{\mathsf{K}} \models^{s_\star} s_1 + s_2$ *if and only if* $\hat{\mathsf{R}}, \hat{\mathsf{K}} \models^{s_\star} s_1$ *and* $\hat{\mathsf{R}}, \hat{\mathsf{K}} \models^{s_\star} s_2$.
- $\hat{\mathsf{R}}, \hat{\mathsf{K}} \models^{s_\star} (n)s$ *if and only if* $\hat{\mathsf{R}}, \hat{\mathsf{K}} \models^{s_\star} s$.
- $\hat{\mathsf{R}}, \hat{\mathsf{K}} \models^{s_\star} [x]^\ell s$ *if and only if* $\hat{\mathsf{R}}, \hat{\mathsf{K}} \models^{s_\star} s$.

Moreover, if $s_1 \equiv s_2$ then $\hat{\mathsf{R}}, \hat{\mathsf{K}} \models^{s_\star} s_1$ if and only if $\hat{\mathsf{R}}, \hat{\mathsf{K}} \models^{s_\star} s_2$.

We are now able to state our subject reduction result. Note that the correctness predicate is only preserved at top-level, that is, when talking about observable computation steps. A computation step is observable, when the substitution of the transition label is empty, that is, when all substitutions induced by a communication have happened.

Theorem 1 (Subject Reduction). *Let s_1 and s_2 be services. If $\hat{\mathsf{R}}, \hat{\mathsf{K}} \models^{s_\star} s_1$ and $s_1 \xrightarrow{p \bullet o \lfloor \emptyset \rfloor \bar{u}^{\ell_i} \bar{n}^{\ell_o}} s_2$ then $\hat{\mathsf{R}}, \hat{\mathsf{K}} \models^{s_\star} s_2$.*

The proof uses Lemma 1 and is given in [1]. It requires an even stronger induction hypothesis than provided by the correctness predicate, because we have to deal with all possible transition labels, in particular with transition labels carrying non-empty substitutions. The stronger induction hypothesis then makes a connection between these substitutions and the analysis result.

The following theorem constitutes the adequacy of our analysis and is proven in [1]. It states that every communication triggering an observed substitution and the substitution itself are in fact recorded in the analysis information.

Theorem 2 (Adequacy). *If $(\hat{\mathsf{R}}, \hat{\mathsf{K}})$ is an acceptable analysis of $s_\star$ and if $s_\star \rightarrow^* s \xrightarrow{p \bullet o \lfloor \emptyset \rfloor \bar{u}^{\ell_o} \bar{n}^{\ell_i}} s'$ then $\langle p o \bar{n} \rangle \in \hat{\mathsf{K}}$ and $\bar{n} \in \Pi_{\bar{u} @ \mathsf{L}.\ell_i}(\hat{\mathsf{R}}.\ell_i)$.*

4.2 Implementation

The basis of our implementation relies on the following Moore family result. It ensures the existence and the uniqueness of a least acceptable analysis result. Its proof is obvious from the syntax directed analysis specification.

Theorem 3 (Moore Family). *For any service s the set of acceptable analysis estimates under $\vdash_{\mathsf{L},\mathsf{F}}$ constitutes a Moore family, i.e.,*

$$\forall A \subseteq \{\hat{\mathsf{R}}, \hat{\mathsf{K}} \mid \hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} s\} : \sqcap A \in \{\hat{\mathsf{R}}, \hat{\mathsf{K}} \mid \hat{\mathsf{R}}, \hat{\mathsf{K}} \vdash_{\mathsf{L},\mathsf{F}} s\}.$$

If we, by a change of perspective, view the analysis specification as logical formulas and acceptable results as models of these formulas, then the Moore family result turns into a model intersection property ensuring a least model corresponding to the least acceptable analysis result.

We have implemented a fully functional prototype in Standard ML generating clauses that lie within the Alternation-free Least Fixed Point (ALFP) fragment of first order logic. Least models of such formulas always exist and can be computed efficiently by, e.g., the Succinct Solver [13,11].

Example 3. The input communication of line (6) of Table 2 gives rise to the following clause:

$$\begin{array}{l}
\forall x_{info}, x_{id}, x_{reply}, x_{sid} : \\
\qquad \hat{\mathsf{R}}.12(x_{info}, x_{id}, x_{reply}, x_{sid}) \land \hat{\mathsf{K}}(\mathsf{p}_s, \mathsf{o}_{confirm}, x_{reply}, x_{info}, x_{sid}) \Rightarrow \\
\qquad\qquad [\ \hat{\mathsf{R}}.13(x_{info}, x_{id}, x_{reply}, x_{sid}) \land \phi \land \psi\ ]
\end{array}$$

This specifies the flow into $\hat{\mathsf{R}}.13$: All variable bindings available at $\hat{\mathsf{R}}.12$ leading to a tuple that may be communicated flow to the continuation at line (7). This corresponds to the X set in rule [RREC]. Formula ϕ in the clause above corresponds to the Y_x set and is left out for brevity. Formula ψ in the clause above corresponds to line (7) of the example representing the analysis of an output according to [RINV] and specifying a flow into $\hat{\mathsf{K}}$

$$\begin{array}{l}
\psi = \forall x_{info}, x_{id}, x_{reply}, x_{sid} : \\
\qquad \hat{\mathsf{R}}.13(x_{info}, x_{id}, x_{reply}, x_{sid}) \land x_{sid} \neq \bot \land x_{info} \neq \bot \land x_{reply} \neq \bot \Rightarrow \\
\qquad\qquad \hat{\mathsf{K}}(\mathsf{p}_s, \mathsf{o}_{Sos}, x_{reply}, x_{info}, x_{sid})
\end{array}$$

□

Complexity. Three quantities determine the complexity of solving the derived ALFP clause:

– the number n of names used in the program,
– the maximal nesting depth of variables, bounded by $m = \max_{\ell \in \mathbf{Lab}} |\mathsf{L}.\ell|$, and
– the maximal length of any sent message, bounded by $k = \max_{u \bullet u' ! \bar{u} \in s} |\bar{u}|$.

The size of the logical universe is bounded by n, while m and n decide the maximal arity of relations. Furthermore, the maximal nesting depth is decided by m, which, by Proposition 1 of [12], results in a complexity bound of $\mathcal{O}(n^{3+k+m})$. This is exponential in the worst case, which is only realised by pathological service specifications, s, where the number of sequenced inputs (m) and/or the arity of sent messages (k) are linear in the size (n) of s. For realistic CWS services, the complexity will be polynomially bound by constants m and k. For the accident service the solution was found in less than a second.

5 Related Work

The separation of scope and binding occurrence of a variable is originally due to Parrow and Victor, who used it to model locally shared memory in the Fusion Calculus [15]. In contrast to CWS, there is only one syntactic category, names, rendering input and output symmetric, i.e., input and output designation can be exchanged while yielding the same fusion.

In [4], the calculus D-Fusion is proposed. It extends the Fusion Calculus with another binder similar to restriction in π-calculus. It retains the symmetry of actions, in fact, it does not even distinguish input and output at all. Apart from this symmetry, D-Fusion is quite close to CWS. However, we prefer to consider CWS as a subset of the COWS [7] calculus, because the latter is more focused on correlations and their use in service-oriented computing.

COWS retains a variant of separation of input and name binding in order to faithfully model correlation. In contrast to Fusion and D-Fusion, inputs and outputs are clearly distinguished, in particular by using two syntactic categories, names and variables, where only substitutions of names for variables are possible. COWS also features pattern matching in input prefixes facilitating correlations. While we, too, embrace these concepts we discard some of the technicalities associated with session management and do not consider the fault and compensation facilities provided by COWS.

To the best of our knowledge [2] the present static analysis is the first static analysis, which is not a type system, developed for the Fusion Calculus or its descendants. Also, it seems to be the first static analysis of correlation. Our analysis is relational in form, which has previously been investigated in the simpler context of the π-calculus [9] based on a reaction semantics. The present development retains the simplicity of the former, even in the context of a more complicated calculus with a labelled structural operational semantics, and testifies to the flexibility of the Flow Logic specification style [14]. In contrast, previous relational approaches [17,5], fashioned within the framework of Abstract Interpretation, have relied on highly customised syntax and semantics and are not easily extended beyond the original context of the π-calculus.

In the context of COWS, Lapadula et. al. use type systems in order to enforce distribution policy annotations [8]. For this approach to work the user has to annotate each piece of data with a region of maximal dissemination (a set of service principals). Static inference combined with a typed semantics, performing appropriate run-time checks, then ensures that the policy is never violated. In contrast, our approach relies neither on user-provided annotations, nor dynamic type-checking. It is fully static and automatically computes a very precise estimate of the data-sets that may reach every single program point. The specified information is very general; hence a region based policy can easily be tested against the computed result. In [8], the complexity of the type system is not discussed, but the general tendency is that checking is polynomial, whereas inference is exponential. In the case of the Succinct Solver, however, there is no complexity gap between checking and inference [13] – both are polynomial for non-pathological specifications.

6 Conclusion

In this paper we have delineated the process calculus CWS for implementing correlation based services in order to focus on the essentials of correlation, e.g., separation of binding and input and pattern matching. We expect that our analysis of CWS transfers easily to other correlation based process calculi. CWS is formulated in a form resembling COWS but lacking its orchestration constructs. In future extensions, we aim at adding these features.

We developed a relational analysis for the CWS process calculus and showed its usefulness for ensuring that service invocations do not interfere in malign ways. We have based our work on a recent relational analysis developed for the π-calculus [9] thereby supporting the claim that the Flow Logic framework facilitates transferring analysis insights between languages (being programming languages or process calculi). While more powerful approaches to relational analysis exist, in particular the work of polyhedral analysis of certain π-processes [17,5], they are substantially harder to transfer to other language because they rely on a special "normal form" for processes established a priori (in case of the π-calculus in the PhD-thesis of [16]).

Despite the guidance offered by the Flow Logic framework and the relational analysis developed for the π-calculus in [9], correlations, i.e., the separation of scope from binding, have presented profound obstacles that we have managed to solve. A key ingredient is the use of $\perp$ to denote the "presence" of a variable that has not yet received its value; this technique is being used in rather deep ways to ensure the semantic correctness of the analysis. The correctness result follows the approach pioneered in [9] in making use of a subject reduction result where "analysability" is not preserved under reduction, whereas the more complex notion of "analysability of all permitted substitution instances" is.

References

1. Bauer, J., Nielson, F., Nielson, H.R., Pilegaard, H.: Relational analysis of correlation. Technical report, Technical University of Munich (2008)
2. Victor, B.: Personal communication (October 2007)

3. Bodei, C., Degano, P., Nielson, F., Nielson, H.R.: Static analysis for the π-calculus with applications to security. Information and Computation 168, 68–92 (2001)
4. Boreale, M., Buscemi, M.G., Montanari, U.: D-fusion: A distinctive fusion calculus. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 296–310. Springer, Heidelberg (2004)
5. Feret, J.: Dependency analysis of mobile systems. In: Le Métayer, D. (ed.) ESOP 2002 and ETAPS 2002. LNCS, vol. 2305, pp. 314–330. Springer, Heidelberg (2002)
6. Guidi, C., Lucchi, R., Gorrieri, R., Busi, N., Zavattaro, G.: SOCK: A calculus for service oriented computing. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 327–338. Springer, Heidelberg (2006)
7. Lapadula, A., Pugliese, R., Tiezzi, F.: A calculus for orchestration of web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, Springer, Heidelberg (2007)
8. Lapadula, A., Pugliese, R., Tiezzi, F.: Regulating data exchange in service oriented applications. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 223–239. Springer, Heidelberg (2007)
9. Nielson, F., Nielson, H.R., Bauer, J., Nielsen, C.R., Pilegaard, H.: Relational analysis for delivery of services. In: Trustworthy Global Computing (to appear, 2007)
10. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, New York (1999)
11. Nielson, F., Nielson, H.R., Sun, H., Buchholtz, M., Hansen, R.R., Pilegaard, H., Seidl, H.: The succinct solver suite. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 251–265. Springer, Heidelberg (2004)
12. Nielson, F., Seidl, H.: Control-flow analysis in cubic time. In: Sands, D. (ed.) ESOP 2001 and ETAPS 2001. LNCS, vol. 2028, pp. 252–268. Springer, Heidelberg (2001)
13. Nielson, F., Seidl, H., Nielson, H.R.: A succinct solver for ALFP. Nord. J. Comput. 9(4), 335–372 (2002)
14. Nielson, H.R., Nielson, F.: Flow Logic: a multi-paradigmatic approach to static analysis. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 223–244. Springer, Heidelberg (2002)
15. Parrow, J., Victor, B.: The fusion calculus: Expressiveness and symmetry in mobile processes. In: LICS, pp. 176–185 (1998)
16. Turner, D.N.: The Polymorphic Pi-calulus: Theory and Implementation. PhD thesis, University of Edinburgh (1996)
17. Venet, A.: Automatic determination of communication topologies in mobile systems. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 152–167. Springer, Heidelberg (1998)

Convex Hull of Arithmetic Automata

Jérôme Leroux

LaBRI, Université de Bordeaux, CNRS
Domaine Universitaire, 351, cours de la Libération,
33405 Talence, France
leroux@labri.fr

Abstract. Arithmetic automata recognize infinite words of digits denoting decompositions of real and integer vectors. These automata are known expressive and efficient enough to represent the whole set of solutions of complex linear constraints combining both integral and real variables. In this paper, the closed convex hull of arithmetic automata is proved rational polyhedral. Moreover an algorithm computing the linear constraints defining these convex set is provided. Such an algorithm is useful for effectively extracting geometrical properties of the whole set of solutions of complex constraints symbolically represented by arithmetic automata.

1 Introduction

The *most significant digit first decomposition* provides a natural way to associate finite words of digits to any integer. Naturally, such a decomposition can be extended to real values just by considering *infinite* words rather than *finite* ones. Intuitively, an infinite word denotes the potentially infinite decimal part of a real number. Last but not least, the most significant digit first decomposition can be extended to real vectors just by interleaving the decomposition of each component into a single infinite word.

Arithmetic automata are Muller automata that recognize infinite words of most significant digit first decompositions of real vectors in a fixed basis of decomposition $r \geq 2$ (for instance $r = 2$ and $r = 10$ are two classical basis of decomposition). Sets symbolically representable by arithmetic automata in basis r are logically characterized [BRW98] as the sets definable in the first order theory $\text{FO}(\mathbb{R}, \mathbb{Z}, +, \leq, X_r)$ where X_r is an additional predicate depending on the basis of decomposition r. In practice, arithmetic automata are usually used for the first order additive theory $\text{FO}(\mathbb{R}, \mathbb{Z}, +, \leq)$ where X_r is discarded. In fact this theory allows to express complex linear constraints combining both integral and real variables that can be represented by particular Muller automata called *deterministic weak Buchi automata* [BJW05]. This subclass of Muller automata has interesting algorithmic properties. In fact, compared to the general class, deterministic weak Buchi automata can be minimized (for the number of states) into a unique canonical form with roughly the same algorithm used for automata recognizing finite words. In particular, these arithmetic automata are well adapted

© Springer-Verlag Berlin Heidelberg 2008

to symbolically represent sets definable in $\mathrm{FO}(\mathbb{R}, \mathbb{Z}, +, \leq)$ obtained after many operations (boolean combinations, quantifications). In fact, since the obtained arithmetic automata only depends on the represented set and not on the potentially long sequence of operations used to compute this set, we avoid unduly complicated arithmetic automata. Intuitively, the automaton minimization algorithm performs like a simplification procedure for $\mathrm{FO}(\mathbb{R}, \mathbb{Z}, +, \leq)$. In particular arithmetic automata are adapted to the symbolic model checking approach computing inductively reachability sets of systems manipulating counters [BLP06] and/or clocks [BH06]. In practice algorithms for effectively computing an arithmetic automaton encoding the solutions of formulas in $\mathrm{FO}(\mathbb{R}, \mathbb{Z}, +, \leq)$ have been recently successfully implemented in tools Lash and Lira [BDEK07]. Unfortunately, interesting qualitative properties are difficult to extract from arithmetic automata. Actually, operations that can be performed on the arithmetic automata computed by tools Lash and Lira are limited to the universality and the emptiness checking (when the set symbolically represented is not empty these tools can also compute a real vector in this set).

Extracting geometrical properties from an arithmetic automaton representing a set $X \subseteq \mathbb{R}^m$ is a complex problem even if X is definable in $\mathrm{FO}(\mathbb{R}, \mathbb{Z}, +, \leq)$. Let us recall related works to this problem. Using a Karr based algorithm [Kar76], the affine hull of X has been proved efficiently computable in polynomial time [Ler04] (even if this result is limited to the special case $X \subseteq \mathbb{N}^m$, it can be easily extended to any arithmetic automata). When $X = \mathbb{Z}^m \cap C$ where C is a rational polyhedral convex set (intuitively when X is equal to the integral solutions of linear constraint systems), it has been proved in [Lat04] that we can effectively compute in exponential time a rational polyhedral convex set C' such that $X = \mathbb{Z}^m \cap C'$. Note that this worst case complexity in theory is not a real problem in practice since the algorithm presented in [Lat04] performs well on automata with more than 100 000 states. In [Lug04] this result was extended to sets $X = F + L$ where F is a finite set of integral vectors and L is a linear set. In [FL05], closed convex hulls of sets $X \subseteq \mathbb{Z}^m$ represented by arithmetic automata are proved rational polyhedral and effectively computable in exponential time. Note that compared to [Lat04], it is not clear that this result can be turn into an efficient algorithm. More recently [Ler05], we provided an algorithm for effectively computing in polynomial time a formula in the Presburger theory $\mathrm{FO}(\mathbb{Z}, +, \leq)$ when $X \subseteq \mathbb{Z}^n$ is Presburger-definable. This algorithm has been successfully implemented in TaPAS [LP08] (The Talence Presburger Arithmetic Suite) and it can be applied on any arithmetic automata encoding a set $X \subseteq \mathbb{Z}^m$ with more than 100 000 states. Actually, the tool decides if an input arithmetic automaton denotes a Presburger-definable set and in this case it returns a formula denoting this set.

In this paper we prove that the closed convex hulls of sets symbolically represented by arithmetic automata are rational polyhedral and effectively computable in exponential time in the worst case. Note that whereas the closed convex hull of a set definable in $\mathrm{FO}(\mathbb{R}, \mathbb{Z}, +, \leq)$ can be easily proved rational polyhedral (thanks to quantification eliminations), it is difficult to prove that the closed convex hulls of arithmetic automata are rational polyhedral. We also provide an algorithm for computing this set. Our algorithm is based on the reduction of the closed convex hull computation to data-flow analysis problems. Note that widening operator is usually used in order to speed up the iterative computation of solutions of such a problem. However, the use of widening operators may lead to loss of precision in the analysis. Our algorithm is based on *acceleration* in convex data-flow analysis [LS07b, LS07a]. Recall that acceleration consists to compute the exact effect of some control-flow cycles in order to speed up the Kleene fix-point iteration.

Outline of the paper : In section 2 the most significant digit first decomposition is extended to any real vector and we introduce the arithmetic automata. In section 3 we provide the closed convex hull computation reduction to (1) a data-flow analysis problem and (2) the computation of the closed convex hull of arithmetic automata representing only decimal values and having a trivial accepting condition. In section 4 we provide an algorithm for computing the closed convex hull of such an arithmetic automaton. Finally in section 5 we prove that the data-flow analysis problem introduced by the reduction can be solved precisely with an accelerated Kleene fix-point iteration algorithm. Due to space limitations, most proofs are only sketched in this paper. A long version of the paper with detailed proofs can be obtained from the author.

2 Arithmetic Automata

This section introduces arithmetic automata (see Fig. 1). These automata recognize infinite words of digits denoting *most significant digit first* decompositions of real and integer vectors.

As usual, we respectively denote by $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ the sets of integers, rationals and real numbers and we denote by $\mathbb{N}, \mathbb{Q}_+, \mathbb{R}_+$ the restrictions of $\mathbb{Z}, \mathbb{Q}, \mathbb{R}$ to the non-negatives. The *components* of an m-dim vector x are denoted by $x[1], \dots, x[m]$.

We first provide some definitions about regular sets of infinite words. We denote by Σ a non-empty finite set called an *alphabet*. An *infinite word* w over Σ is a function $w \in \mathbb{N} \to \Sigma$ defined over $\mathbb{N} \backslash \{0\}$ and a *finite word* σ over Σ is a function $\sigma \in \mathbb{N} \to \Sigma$ defined over a set $\{1, \dots, k\}$ where $k \in \mathbb{N}$ is called the *length* of σ and denoted by $|\sigma|$. *In this paper, a finite word over* Σ *is denoted by* σ *with some subscript indices and an infinite word over* Σ *is denoted by* w. As usual Σ^* and Σ^ω respectively denote the set of finite words and the set of infinite words over Σ. The concatenation of two finite words $\sigma_1, \sigma_2 \in \Sigma^*$ and the concatenation of a finite word $\sigma \in \Sigma^*$ with an infinite word $w \in \Sigma^\omega$ are denoted by $\sigma_1\sigma_2$ and σw. A *graph labelled* by Σ is a tuple $G = (Q, \Sigma, T)$ where Q is a non empty finite set of *states* and $T \subseteq Q \times \Sigma \times Q$ is a set of *transitions*. A *finite path* π in a graph G is a finite word $\pi = t_1 \dots t_k$ of $k \geq 0$ transitions $t_i \in T$ such that there exists a sequence $q_0, \dots, q_k \in Q$ and a sequence $a_1, \dots, a_k \in \Sigma$ such that $t_i = (q_{i-1}, a_i, q_i)$ for any $1 \leq i \leq k$. The finite word $\sigma = a_1 \dots a_k$ is called the *label* of π and such a path π is also denoted by $q_0 \xrightarrow{\sigma} q_k$ or just $q_0 \rightarrow q_k$. We also say that π is a path *starting* from q_0 and *terminating* in q_k. When $q_0 = q_k$ and $k \geq 1$, the path π is called a cycle on q_0. Such a cycle is said *simple* if the states $q_0, \dots, q_{k-1}$ are distinct. Given an integer $m \geq 1$, a graph G is called an m-graph if m divides the length of any cycle in G. An *infinite path* θ is an infinite word of transitions such that any prefixes $\pi_k = \theta(1) \dots \theta(k)$ is a finite path. The unique infinite word $w \in \Sigma^\omega$ such that $\sigma_k = w(1) \dots w(k)$ is the label of the finite path π_k for any $k \in \mathbb{N}$ is called the *label* of θ. We say that θ is *starting* from q_0 if q_0 is the unique state such that any prefix of θ is starting from q_0. *In the sequel, a finite path is denoted by* π *and an infinite path is denoted by* θ. The *set of infinite paths starting from* q_0 is naturally denoted with the capital letter $\Theta_G(q_0)$. The set F of states $q \in Q$ such that there exists an infinite number of prefix of θ terminating in q is called the set of *states visited infinitely often* by θ. Such a path is denoted by $q_0 \xrightarrow{w} F$ or just $q_0 \rightarrow F$. A *Muller automaton* A is a tuple $A = (Q, \Sigma, T, Q_0, \mathcal{F})$ where (Q, Σ, T) is a graph, $Q_0 \subseteq Q$ is the *initial condition* and $\mathcal{F} \subseteq \mathcal{P}(Q)$ is the *accepting condition*. The language $L(A) \subseteq \Sigma^\omega$ *recognized* by a Muller automaton A is the set of infinite words $w \in \Sigma^\omega$ such that there exists an infinite path $q_0 \xrightarrow{w} F$ with $q_0 \in Q_0$ and $F \in \mathcal{F}$.

Now, we introduce the *most significant digit first decomposition* of real vectors. In the sequel $m \geq 1$ is an integer called *the dimension*, $r \geq 2$ is an integer called the *basis of decomposition*, $\Sigma_r = \{0, \dots, r-1\}$ is called the alphabet of *r-digits*, and $S_r = \{0, r-1\}$ is called the alphabet of *sign r-digits*. The most significant r-digit first decomposition provides a natural way to associate to any real vector $x \in \mathbb{R}^m$ a tuple $(s, \sigma, w) \in S_r^m \times (\Sigma_r^m)^* \times \Sigma_r^\omega$. Intuitively (s, σ) and w are respectively associated to an integer vector $z \in \mathbb{Z}^m$ and a decimal vector

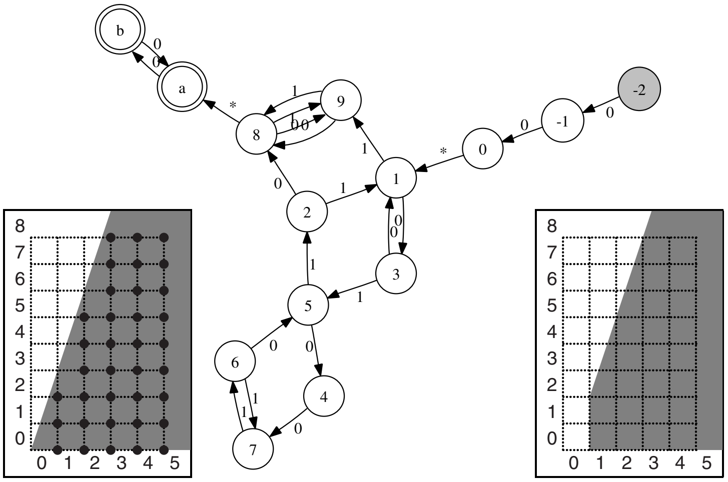


Fig. 1. On the left, the rational polyhedral convex set $C = \{x \in \mathbb{R}^2 \mid 3x[1] > x[2] \wedge x[2] \geq 0\}$ in gray and the set $X = \mathbb{Z}^2 \cap C$ of integers depicted by black bullets. On the center, an arithmetic automaton symbolically representing X in basis 2. On the right, the closed convex hull of X equals to $\mathrm{cl} \circ \mathrm{conv}(X) = \{x \in \mathbb{R}^2 \mid 3x[1] \geq x[2] + 1 \wedge x[2] \geq 0 \wedge x[1] \geq 1\}$ represented in gray.

$d \in [0,1]^m$ satisfying $x = z + d$. Moreover, $s[i] = 0$ corresponds to $z[i] \geq 0$ and $s[i] = r-1$ corresponds to $z[i] < 0$. More formally, a *most significant r-digit first decomposition* of a real vector $x \in \mathbb{R}^m$ is a tuple $(s, \sigma, w) \in S_r^m \times (\Sigma_r^m)^* \times \Sigma_r^\omega$ such that for any $1 \leq i \leq m$, we have:

$$x[i] = r^{\frac{|\sigma|}{m}} \frac{s(i)}{1-r} + \sum_{j=0}^{\frac{|\sigma|}{m}-1} r^j \sigma(mj+i) + \sum_{j=0}^{+\infty} \frac{w(mj+i)}{r^{j+1}}$$

The previous equality is divided in two parts by introducing the functions $\lambda_{r,m} \in \Sigma_r^\omega \to [-1,0]^m$ and $\gamma_{r,m} \in S_r^m \times (\Sigma_r^m)^* \to \mathbb{Z}^m$ defined for any $1 \leq i \leq m$ by the following equalities. Note the sign in front of the definition of $\lambda_{r,m}$. This sign simplifies the presentation of this paper and it is motivated in the sequel.

$$-\lambda_{r,m}(w)[i] = \sum_{j=0}^{+\infty} \frac{w(mj+i)}{r^{j+1}}$$

$$\gamma_{r,m}(s,\sigma)[i] = r^{\frac{|\sigma|}{m}} \frac{s(i)}{1-r} + \sum_{j=0}^{\frac{|\sigma|}{m}-1} r^j \sigma(mj+i)$$

Definition 2.1 ([BRW98]). *An* arithmetic automaton *A in basis r and in dimension m is a Muller automaton over the alphabet $\Sigma_r \cup \{\star\}$ that recognizes a language $L \subseteq S_r^m \star (\Sigma_r^m)^* \star \Sigma_r^\omega$. The following set $X \subseteq \mathbb{R}^m$ is called the set symbolically represented by A:*

$$X = \{\gamma_{r,m}(s,\sigma) - \lambda_{r,m}(w) \mid s \star \sigma \star w \in L\}$$

Example 2.2. The arithmetic automaton depicted in Fig. 1 symbolically represents $X = \{x \in \mathbb{N}^2 \mid 3x[1] > x[2]\}$. This automaton has been obtained automatically from the tool Lash through the tool-suite TaPAS[LP08].

We observe that *Real Vector Automata (RVA)* and *Number Decision Diagrams (NDD)* [BRW98] are particular classes of arithmetic automata. In fact, RVA and NDD are arithmetic automata A that symbolically represent sets X included respectively in $\mathbb{R}^m$ and $\mathbb{Z}^m$ and such that the accepted languages $L(A)$ satisfy:

$$L(A) = \{s \star \sigma \star w \mid \gamma_{r,m}(s,\sigma) - \lambda_{r,m}(w) \in X\} \qquad \text{if } A \text{ is a RVA}$$
$$L(A) = \{s \star \sigma \star 0^\omega \mid \gamma_{r,m}(s,\sigma) \in X\} \qquad \text{if } A \text{ is a NDD}$$

Since in general a NDD is not a RVA and conversely a RVA is not a NDD, we consider arithmetic automata in order to solve the closed convex hull computation uniformly for these two classes. Note that simple (even if computationally expensive) automata transformations show that sets symbolically representable by arithmetic automata in basis r are exactly the sets symbolically representable by RVA in basis r. In particular [BRW98], sets symbolically representable by arithmetic automata in basis r are exactly the sets definable in FO$(\mathbb{R}, \mathbb{Z}, +, \leq, X_r)$ where $X_r \subseteq \mathbb{R}^3$ is a basis dependant predicate defined in [BRW98]. This characterization shows that arithmetic automata can symbolically represent sets of solutions of complex linear constraints combining both integral and real values. Recall that the construction of arithmetic automata from formulae in $\text{FO}(\mathbb{R}, \mathbb{Z}, +, \leq, X_r)$ is effective and tools LASH and LIRA [BDEK07] implement efficient algorithms for the restricted logic $\text{FO}(\mathbb{R}, \mathbb{Z}, +, \leq)$. The predicate X_r is discarded in these tools in order to obtain arithmetic automata that are deterministic weak Buchi automata [BJW05]. In fact these automata have interesting algorithmic properties (minimization and deterministic form).

3 Reduction to Data-Flow Analysis Problems

In this section we reduce the computation of the closed convex hull of sets symbolically represented by arithmetic automata to data-flow analysis problems.

We first recall some general notions about complete lattices. Recall that a *complete lattice* is any partially ordered set $(A, \sqsubseteq)$ such that every subset $X \subseteq A$ has a *least upper bound* $\bigsqcup X$ and a *greatest lower bound* $\bigsqcap X$. The *supremum* $\bigsqcup A$ and the *infimum* $\bigsqcap A$ are respectively denoted by $\top$ and $\bot$. A function $f \in A \rightarrow A$ is *monotonic* if $f(x) \sqsubseteq f(y)$ for all $x \sqsubseteq y$ in A. For any complete lattice $(A, \sqsubseteq)$ and any set Q, we also denote by $\sqsubseteq$ the partial order on $Q \rightarrow A$ defined as the point-wise extension of $\sqsubseteq$, i.e. $f \sqsubseteq g$ iff $f(q) \sqsubseteq g(q)$ for all $q \in Q$. The partially ordered set $(Q \rightarrow A, \sqsubseteq)$ is also a complete lattice, with lub $\bigsqcup$ and glb $\bigsqcap$ satisfying $(\bigsqcup F)(s) = \bigsqcup\{f(s) \mid f \in F\}$ and $(\bigsqcap F)(s) = \bigsqcap\{f(s) \mid f \in F\}$ for any subset $F \subseteq Q \rightarrow A$.

Now, we recall notions about the complete lattice of closed convex sets. A function $f \in \mathbb{R}^n \rightarrow \mathbb{R}^m$ is said *linear* if there exists a sequence $(M_{i,j})_{i,j}$ of reals indexed by $1 \leq i \leq m$ and $1 \leq j \leq n$ and a sequence $(v_i)_i$ of reals indexed by $1 \leq i \leq m$ such that $f(x)[i] = \sum_{j=1}^{n} M_{i,j}x[j] + v_i$ for any $x \in \mathbb{R}^n$ and for any $1 \leq i \leq m$. When the coefficients $(M_{i,j})_{i,j}$ and $(v_i)_i$ are rational, the linear function f is said *rational*. The function $f' \in \mathbb{R}^m \rightarrow \mathbb{R}^n$ defined by $f'(x)[i] = \sum_{j=1}^{n} M_{i,j}x[j]$ for any $x \in \mathbb{R}^n$ and for any $1 \leq i \leq m$ is called the *uniform form* of f. A set $R \subseteq \mathbb{R}^m$ is said *closed* if the limit of any convergent sequence of vectors in R is in R. Recall that any set $X \subseteq \mathbb{R}^m$ is included in a minimal for the inclusion closed set. This closed set is called the *topological closure* of X and it is denoted by $\text{cl}(X)$. Let us recall some notions about convex sets (for more details, see [Sch87]). A *convex combination* of $k \geq 1$ vectors $x_1, \ldots, x_k \in \mathbb{R}^m$ is a vector x such that there exists $r_1, \ldots, r_k \in \mathbb{R}_+$ satisfying $r_1 + \cdots + r_k = 1$ and $x = r_1x_1 + \cdots + r_kx_k$. A set $C \subseteq \mathbb{R}^m$ is said *convex* if any convex combination of vectors in C is in C. Recall that any $X \subseteq \mathbb{R}^m$ is included in a minimal for the inclusion convex set. This convex set is called the *convex hull* of X and it is denoted by $\text{conv}(X)$. A convex set $C \subseteq \mathbb{R}^m$ is said *rational polyhedral* if there exists a rational linear function $f \in \mathbb{R}^m \rightarrow \mathbb{R}^n$ such that C is the set of vectors $x \in \mathbb{R}^m$ such that $\bigwedge_{i=1}^{n} f(x)[i] \leq 0$. Recall that $\text{cl}(\text{conv}(X)) = \text{conv}(\text{cl}(X))$, $\text{cl}(f(X)) = f(\text{cl}(X))$ and $\text{conv}(f(X)) = f(\text{conv}(X))$ for any $X \subseteq \mathbb{R}^m$ and for any linear function $f \in \mathbb{R}^m \rightarrow \mathbb{R}^n$. The class of closed convex subsets of $\mathbb{R}^m$ is written $\mathcal{C}_m$. We denote by $\sqsubseteq$ the inclusion partial order on $\mathcal{C}_m$. Observe that $(\mathcal{C}_m, \sqsubseteq)$ is a complete lattice, with lub $\bigsqcup$ and glb $\bigsqcap$ satisfying $\bigsqcup \mathcal{C} = \mathrm{cl} \circ \mathrm{conv}(\bigcup \mathcal{C})$ and $\bigsqcap \mathcal{C} = \bigcap \mathcal{C}$ for any subset $\mathcal{C} \subseteq \mathcal{C}_m$.

Example 3.1. Let $X = \mathbb{Z}^2 \cap C$ where C is the convex set $C = \{x \in \mathbb{R}^2 \mid 3x[1] > x[2] \wedge x[2] \geq 0\}$ (see Fig. 1). Observe that $\mathrm{cl} \circ \mathrm{conv}(X) = \{x \in \mathbb{R}^2 \mid 3x[1] \geq x[2] + 1 \wedge x[2] \geq 0 \wedge x[1] \geq 1\}$ is strictly included in C.

In the previous section, we introduced two functions $\lambda_{r,m}$ and $\gamma_{r,m}$. Intuitively these functions "compute" respectively decimal vectors associated to infinite words and integer vectors associated to finite words equipped with sign vectors. We now introduce two functions $\Lambda_{r,m,\sigma}$ and $\Gamma_{r,m,\sigma}$ that "*partially compute*" the same vectors than $\lambda_{r,m}$ and $\gamma_{r,m}$. More formally, let us consider the unique sequences $(\Lambda_{r,m,\sigma})_{\sigma \in \Sigma_r^*}$ and $(\Gamma_{r,m,\sigma})_{\sigma \in \Sigma_r^*}$ of linear functions $\Lambda_{r,m,\sigma}, \Gamma_{r,m,\sigma} \in \mathbb{R}^m \rightarrow \mathbb{R}^m$ inverse of each other and satisfying $\Lambda_{r,m,\sigma_1\sigma_2} = \Lambda_{r,m,\sigma_1} \circ \Lambda_{r,m,\sigma_2}$, $\Gamma_{r,m,\sigma_1\sigma_2} = \Gamma_{r,m,\sigma_2} \circ \Gamma_{r,m,\sigma_1}$ for any $\sigma_1, \sigma_2 \in \Sigma_r^*$, such that $\Lambda_{r,m,\epsilon}$ and $\Gamma_{r,m,\epsilon}$ are the identity function and such that $\Lambda_{r,m,a}$ and $\Gamma_{r,m,a}$ with $a \in \Sigma_r$ satisfy the following equalities where $x \in \mathbb{R}^m$:

$$\Lambda_{r,m,a}(x) = (\frac{x[m] - a}{r}, x[1], \ldots, x[m-1])$$
$$\Gamma_{r,m,a}(x) = (x[2], \ldots, x[m], rx[1] + a)$$

We first prove the following two equalities (1) and (2) that explain the link between the notations $\lambda_{r,m}$ and $\gamma_{r,m}$ and their capital forms $\Lambda_{r,m,\sigma}$ and $\Gamma_{r,m,\sigma}$. Observe that $\Lambda_{r,m,a}(\lambda_{r,m}(w)) = \lambda_{r,m}(aw)$ for any $a \in \Sigma_r$ and for any $w \in \Sigma_r^\omega$. An immediate induction over the length of $\sigma \in \Sigma_r^*$ provides equality (1). Note also that $\Gamma_{r,m,a_1 \ldots a_m}(x) = rx + (a_1, \ldots, a_m)$ for any $a_1, \ldots, a_m \in \Sigma_r$. Thus an immediate induction provides equality (2).

$$\lambda_{r,m}(\sigma w) = \Lambda_{r,m,\sigma}(\lambda_{r,m}(w)) \qquad \forall \sigma \in \Sigma_r^* \quad \forall w \in \Sigma_r^\omega \tag{1}$$
$$\gamma_{r,m}(s, \sigma) = \Gamma_{r,m,\sigma}(\frac{s}{1-r}) \qquad \forall \sigma \in (\Sigma_r^m)^* \quad \forall s \in S_r^m \tag{2}$$

We now reduce the computation of the closed convex hull C of a set $X \subseteq \mathbb{R}^m$ represented by an arithmetic automaton $A = (Q, \Sigma, T, Q_0, \mathcal{F})$ in basis r to data-flow analysis problems. We can assume w.l.o.g that (Q, Σ, T) is a m-graph. As the language recognized by A is included in $S_r^m \star (\Sigma_r^m)^* \star \Sigma_r^\omega$, the set of states can be partitioned into sets depending intuitively on the number of occurrences $|\sigma|_\star$ of the $\star$ symbol in a word $\sigma \in \Sigma^*$. More formally, we consider the set Q_S of states reading *signs*, the set Q_I reading *integers*, and the set Q_D reading *decimals* defined by:

$$Q_S = \{q \in Q \mid \exists (q_0, \sigma, F) \in Q_0 \times \Sigma^* \times \mathcal{F} \quad |\sigma|_\star = 0 \quad \wedge \quad q_0 \xrightarrow{\sigma} q \rightarrow F\}$$
$$Q_I = \{q \in Q \mid \exists (q_0, \sigma, F) \in Q_0 \times \Sigma^* \times \mathcal{F} \quad |\sigma|_\star = 1 \quad \wedge \quad q_0 \xrightarrow{\sigma} q \rightarrow F\}$$
$$Q_D = \{q \in Q \mid \exists (q_0, \sigma, F) \in Q_0 \times \Sigma^* \times \mathcal{F} \quad |\sigma|_\star = 2 \quad \wedge \quad q_0 \xrightarrow{\sigma} q \rightarrow F\}$$

We also consider the m-graphs G_S, G_I and G_D obtained by restricting G respectively to the states Q_S, Q_I and Q_D and formally defined by:

$$\begin{aligned} G_S &= (Q_S, \Sigma_r, T_S) && \text{with } T_S = T \cap (Q_s \times \Sigma_r \times Q_S) \\ G_I &= (Q_I, \Sigma_r, T_I) && \text{with } T_I = T \cap (Q_I \times \Sigma_r \times Q_I) \\ G_D &= (Q_D, \Sigma_r, T_D) && \text{with } T_D = T \cap (Q_D \times \Sigma_r \times Q_D) \end{aligned}$$

Example 3.2. $Q_S = \{-2, -1, 0\}$, $Q_I = \{1, \dots, 9\}$ and $Q_D = \{a, b\}$ in Fig. 1.

The closed convex hull $C = \mathrm{cl} \circ \mathrm{conv}(X)$ is obtained from the valuations $C_I \in Q_I \to \mathcal{C}_m$ and $C_D \in Q_D \to \mathcal{C}_m$ defined by $C_I = \mathrm{cl} \circ \mathrm{conv}(X_I)$ and $C_D = \mathrm{cl} \circ \mathrm{conv}(X_D)$ where X_I and X_D are given by:

$$\begin{aligned} X_I(q_I) &= \{\Gamma_{r,m,\sigma}(\frac{s}{1-r}) \mid s \in S_r^m \quad \sigma \in \Sigma_r^* \quad \exists q_0 \in Q_0 \quad q_0 \xrightarrow{s\star\sigma} q_I\} \\ X_D(q_D) &= \{\lambda_{r,m}(w) \mid w \in \Sigma_r^\omega \quad \exists F \in \mathcal{F} \quad q_D \xrightarrow{w} F\} \end{aligned}$$

In fact from the definition of arithmetic automata we get:

$$C = \bigsqcup_{\substack{(q_I, q_D) \in Q_I \times Q_D \\ (q_I, \star, q_D) \in T}} C_I(q_I) - C_D(q_D)$$

We now provide data-flow analysis problems whose C_I and C_D are solutions. Observe that m-graphs naturally denote control-flow graphs. Before associating semantics to m-graph transitions, we first show that C_I and C_D are some fix-point solutions. As $\mathrm{cl} \circ \mathrm{conv}$ and $\Gamma_{r,m,a}$ are commutative, from the inclusion $\Gamma_{r,m,a}(X_I(q_1)) \subseteq X_I(q_2)$ we deduce that C_I satisfies the relation $\Gamma_{r,m,a}(C_I(q_1)) \sqsubseteq C_I(q_2)$ for any transition $(q_2, a, q_2) \in T_I$. Symmetrically, as $\mathrm{cl} \circ \mathrm{conv}$ and $\Lambda_{r,m,a}$ are commutative, from the inclusion $\Lambda_{r,m,a}(X_D(q_2)) \subseteq X_D(q_1)$, we deduce that $\Lambda_{r,m,a}(C_D(q_2)) \sqsubseteq C_D(q_1)$ for any transition $(q_1, a, q_2) \in T_D$. Intuitively C_I and C_D are two fix-point solutions of different systems. More formally, we associate two distinct semantics to a transition $t = (q_1, a, q_2)$ of a m-graph $G = (Q, \Sigma_r, T)$ by considering the monotonic functions $\Lambda_{G,m,t}$ and $\Gamma_{G,m,t}$ over the complete lattice $(Q \to \mathcal{C}_m, \sqsubseteq)$ defined for any $C \in Q \to \mathcal{C}_m$ and for any $q \in Q$ by the following equalities:

$$\Lambda_{G,m,t}(C)(q) = \begin{cases} \Lambda_{r,m,a}(C(q_2)) & \text{if } q = q_1 \\ C(q) & \text{if } q \neq q_1 \end{cases}$$

$$\Gamma_{G,m,t}(C)(q) = \begin{cases} \Gamma_{r,m,a}(C(q_1)) & \text{if } q = q_2 \\ C(q) & \text{if } q \neq q_2 \end{cases}$$

Observe that C_D is a fix-point solution of the data-flow problem $\Lambda_{G_D,m,t}(C_D) \sqsubseteq C_D$ for any transition $t \in T_D$ and C_I is a fix-point solution of the data-flow problem $\Gamma_{G_I,m,t}(C_I) \sqsubseteq C_I$ for any transition $t \in T_I$. In the next sections 3.1 and 3.2 we show that C_D and C_I can be characterized by these two data-flow analysis problems.

3.1 Reduction for C_D

The computation of C_D is reduced to a data-flow analysis problem for the m-graph G_D equipped with the semantics $(\Lambda_{G_D,m,t})_{t\in T_D}$.

Given an infinite path θ labelled by w, we denote by $\lambda_{r,m}(\theta)$ the vector $\lambda_{r,m}(w)$. Given a m-graph G labelled by Σ_r, we denote by $\Lambda_{G,m}$, the valuation $\mathrm{cl}\circ\mathrm{conv}(\lambda_{r,m}(\Theta_G))$ (recall that $\Theta_G(q)$ denotes the set of infinite paths starting from q). This notation is motivated by the following Proposition 3.3.

Proposition 3.3. *The valuation $\Lambda_{G,m}$ is the unique minimal valuation $C \in Q \rightarrow \mathcal{C}_m$ such that $\Lambda_{G,m,t}(C) \sqsubseteq C$ for any transition $t \in T$ and such that $C(q) \neq \emptyset$ for any state $q \in Q$ satisfying $\Theta_G(q) \neq \emptyset$.*

The following Proposition 3.4 provides the reduction.

Proposition 3.4. $C_D = \Lambda_{G_D,m}$

Proof. We have previously proved that $\Lambda_{G_D,m,t}(C_D) \sqsubseteq C_D$ for any transition $t \in T_D$. Moreover, as $C_D(q_D) \neq \emptyset$ for any $q_D \in Q_D$, we deduce the relation $\Lambda_{G_D,m} \sqsubseteq C_D$ by minimality of $\Lambda_{G_D,m}$. For the other relation, just observe that $X_D \subseteq \lambda_{r,m}(\Theta_{G_D})$ and apply $\mathrm{cl}\circ\mathrm{conv}$. □

3.2 Reduction for C_I

The computation of C_I is reduced to data-flow analysis problems for the m-graphs G_S and G_I respectively equipped with the semantics $(\Gamma_{G_S,m,t})_{t\in T_S}$ and $(\Gamma_{G_I,m,t})_{t\in T_I}$.

Given a m-graph $G = (Q, \Sigma_r, T)$ and an *initial valuation* $C_0 \in Q \rightarrow \mathcal{C}_m$, it is well-known from Knaster-Tarski's theorem that there exists a unique minimal valuation $C \in Q \rightarrow \mathcal{C}_m$ such that $C_0 \sqsubseteq C$ and $\Gamma_{G,m,t}(C) \sqsubseteq C$ for any $t \in T$. We denote by $\Gamma_{G,m}(C_0)$ this unique valuation.

Symmetrically to the definitions of C_I and C_D we also consider the valuation $C_S \in Q_S \rightarrow \mathcal{C}_m$ defined by $C_S = \mathrm{cl}\circ\mathrm{conv}(X_S)$ where X_S is given by:

$$X_S(q_S) = \{\Gamma_{r,m,s}(0,\ldots,0) \mid s \in S_r^* \quad \exists q_0 \in Q_0 \quad q_0 \xrightarrow{s} q_S\}$$

The reduction comes from the following Proposition 3.5 where $C_{S,0} \in Q_S \rightarrow \mathcal{C}_m$ and $C_{I,0} \in Q_I \rightarrow \mathcal{C}_m$ are the following two initial valuations:

$$C_{S,0}(q_S) = \begin{cases} \emptyset & \text{if } q_S \notin Q_0 \\ \{(0,\ldots,0)\} & \text{if } q_S \in Q_0 \end{cases}$$

$$C_{I,0}(q_I) = \frac{1}{1-r} \bigsqcup_{\substack{q_S\in Q_S \\ (q_S,\star,q_I)\in T}} C_S(q_S)$$

Proposition 3.5. $C_S = \Gamma_{G_S,m}(C_{S,0})$ *and* $C_I = \Gamma_{G_I,m}(C_{I,0})$.

Proof. First observe that $X_S \subseteq \Gamma_{G_S,m}(C_{S,0})$ and $X_I \subseteq \Gamma_{G_I,m}(C_{I,0})$. Thus $C_S \sqsubseteq \Gamma_{G_S,m}(C_{S,0})$ and $C_I \sqsubseteq \Gamma_{G_I,m}(C_{I,0})$ by applying $\mathrm{cl}\circ\mathrm{conv}$. Finally, as $\Gamma_{r,m,a}$ and cl ∘ conv are commutative, we deduce that $\Gamma_{G_S,m,t}(C_S) \sqsubseteq C_S$ for any $t \in T_S$ and $\Gamma_{G_I,m,t}(C_I) \sqsubseteq C_I$ for any $t \in T_I$. The minimality of $\Gamma_{G_S,m}(C_{S,0})$ and $\Gamma_{G_I,m}(C_{I,0})$ provide $\Gamma_{G_S,m}(C_{S,0}) \sqsubseteq C_S$ and $\Gamma_{G_I,m}(C_{I,0}) \sqsubseteq C_I$. □

4 Infinite Paths Convex Hulls

In this section $G = (Q, \Sigma_r, T)$ is a m-graph. We prove that $\Lambda_{G,m}(q)$ is equal to the convex hull of a finite set of rational vectors. Moreover, we provide an algorithm for computing the minimal sets $\Lambda^0_{G,m}(q) \subseteq \mathbb{Q}^m$ for every $q \in Q$ such that $\Lambda_{G,m} = \mathrm{conv}(\Lambda^0_{G,m})$ in exponential time in the worst case.

A *fry-pan* θ in a graph G is an infinite path $\theta = t_1 \dots t_i (t_{i+1} \dots t_k)^\omega$ where $0 \leq i < k$ and where $t_1 = (q_0 \to q_1), \dots t_k = (q_{k-1} \to q_k)$ are transitions such that $q_k = q_i$. A fry-pan is said *simple* if $q_0, \dots, q_{k-1}$ are distinct states. The *finite set of simple fry-pans* starting from q is denoted by $\Theta^S_G(q)$. As expected, we are going to prove that $\Lambda_{G,m} = \mathrm{conv}(\lambda_{r,m}(\Theta^S_G))$ and $\lambda_{r,m}(\Theta^S_G(q)) \subseteq \mathbb{Q}^m$.

We first prove that $\lambda_{r,m}(\theta)$ is rational for any fry-pan θ. Given $\sigma \in \Sigma_r^+$, the following Lemma 4.1 shows that $\lambda_{r,m}(\sigma^\omega)$ is the unique solution of the rational linear system $\Lambda_{r,m,\sigma}(x) = x$. In particular $\lambda_{r,m}(\sigma^\omega)$ is a rational vector. From equality (1) given in page 53, we deduce that the vector $\lambda_{r,m}(\theta)$ is rational for any fry-pan θ.

Lemma 4.1. *$\lambda_{r,m}(\sigma^\omega)$ is the unique fix-point of $\Lambda_{r,m,\sigma}$ for any $\sigma \in \Sigma_r^+$.*

The following Proposition 4.2 (see the graphical support given in Fig. 2) is used in the sequel for effectively computing $\Lambda_{G,m}$ thanks to a fix-point iteration algorithm.

Proposition 4.2. *Let $t = (q, a, q')$ be a transition and let θ' be a simple fry-pan starting from q' such that the fry-pan $t\theta'$ is not simple. In this case there exists a minimal non-empty prefix π of $t\theta'$ terminating in q. Moreover the fry-pan θ such that $t\theta' = \pi\theta$ and the fry-pan π^ω are simple and such that $\Lambda_{r,m,a}(\lambda_{r,m}(\theta')) \in \mathrm{conv}(\{\lambda_{r,m}(\theta), \lambda_{r,m}(\pi^\omega)\})$.*

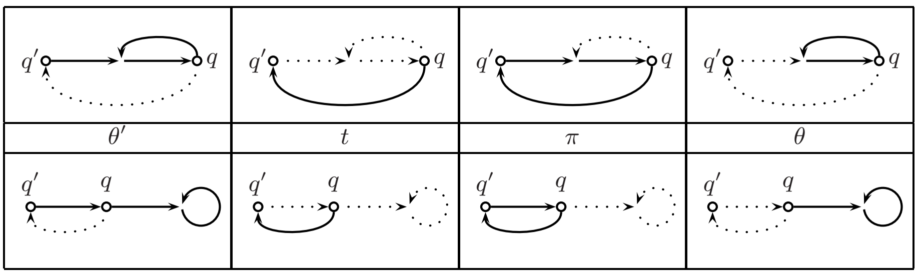


Fig. 2. A graphical support for Proposition 4.2 where θ' denotes a simple fry-pan starting from a state q' and $t = (q, a, q')$ is a transition such that the fry-pan $t\theta'$ is not simple. That means the state q is visited by θ'. Note that q is visited either once or infinitely often. These two situations are depicted respectively on the top line and the bottom line of the tabular.

Proof. As $t\theta'$ is not simple whereas θ' is simple we deduce that there exists a decomposition of $t\theta'$ into $\pi\theta$ where π is the minimal non-empty prefix of $t\theta'$ terminating in q. Let π be the non empty path with the minimal length. Observe that π is a simple cycle and thus π^ω is a simple fry-pan. Moreover, as θ is a suffix of the simple fry-pan θ', we also deduce that θ is a simple fry-pan. Observe that $\lambda_{r,m}(t\theta') = \lambda_{r,m}(\pi\theta)$. Moreover, as π is a cycle in a m-graph we deduce that m divides its length. Denoting by σ the label of π, we deduce that $\sigma \in (\Sigma_r^m)^+$. Now, observe that $\Lambda_{r,m,\sigma}(x) = (1 - r^{-\frac{|\sigma|}{m}})\lambda_{r,m}(\sigma^\omega) + r^{-\frac{|\sigma|}{m}}x$ for any $x \in \mathbb{R}^m$. We deduce that $\Lambda_{r,m,a}(\lambda_{r,m}(\theta')) = (1 - r^{-\frac{|\sigma|}{m}})\lambda_{r,m}(\pi^\omega) + r^{-\frac{|\sigma|}{m}}\lambda_{r,m}(\theta)$. Thus $\Lambda_{r,m,a}(\lambda_{r,m}(\theta')) \in \mathrm{conv}(\{\lambda_{r,m}(\theta), \lambda_{r,m}(\pi^\omega)\})$. □

From the previous Proposition 4.2 we deduce the following Proposition 4.3.

Proposition 4.3. *We have* $\Lambda_{G,m} = \mathrm{conv}(\lambda_{r,m}(\Theta_G^S))$.

We deduce that there exists a minimal finite set $\Lambda^0_{G,m}(q) \subseteq \mathbb{Q}^m$ such that $\Lambda_{G,m} = \mathrm{conv}(\Lambda^0_{G,m})$. Note that an exhaustive computation of the whole set $\Theta_G^S(q)$ provides the set $\Lambda^0_{G,m}(q)$ by removing vectors that are convex combination of others. The efficiency of such an algorithm can be greatly improved by computing inductively subsets $\Theta(q) \subseteq \Theta_G^S(q)$ and get rid of any fry-pan $\theta \in \Theta(q)$ as soon as it becomes a convex combination of other fry-pans in $\Theta(q)\backslash\{\theta\}$. The algorithm Cycle is based on this idea.

Corollary 4.4. *The algorithm Cycle(G,m) terminates by iterating the main while loop at most* $|T|^{|Q|}$ *times and it returns* $\Lambda^0_{G,m}$.

```
Cycle($G = (Q, \Sigma_r, T)$ be a $m-$graph, $m \in \mathbb{N}\backslash\{0\}$)
   for each state $q \in Q$
      if $\Theta_G^S(q) \neq \emptyset$
         let $\theta \in \Theta_G^S(q)$
      let $\Theta(q) \leftarrow \{\theta\}$
      else
         let $\Theta(q) \leftarrow \emptyset$
   while there exists $t = (q, a, q') \in T$ and $\theta' \in \Theta(q')$
      such that $\Lambda_{r,m,a}(\lambda_{r,m}(\theta')) \notin \mathrm{conv}(\lambda_{r,m}(\Theta(q)))$
      if $t\theta'$ is simple
         let $\Theta(q) \leftarrow \Theta(q) \cup \{t\theta'\}$
      else
         let $\pi$ be the minimal strict prefix of $t\theta'$ terminating in $q$
         let $\theta$ be such that $t\theta' = \pi\theta$
      let $\Theta(q) \leftarrow \Theta(q) \cup \{\theta, \pi^\omega\}$
      while there exists $\theta_0 \in \Theta(q)$
         such that $\mathrm{conv}(\lambda_{r,m}(\Theta(q))) = \mathrm{conv}(\lambda_{r,m}(\Theta(q)\backslash\{\theta_0\}))$
         let $\Theta(q) \leftarrow \Theta(q)\backslash\{\theta_0\}$
   return $\lambda_{r,m}(\Theta)$                              //$\Lambda^0_{G,m}$
```

5 Fix-Point Computation

In this section we prove that the minimal post-fix-point $\varGamma_{G,m}(C_0)$ is effectively rational polyhedral for any m-graph $G=(Q,\varSigma_r,T)$ and for any rational polyhedral initial valuation $C_0 \in Q \rightarrow \mathcal{C}_m$. We deduce that the closed convex hull of sets symbolically represented by arithmetic automata are effectively rational polyhedral.

Example 5.1. Let $m=1$ and $G=(\{q\},\varSigma_r,\{t\})$ where $t=(q,r-1,q)$ and $C_0(q)=\{0\}$. Observe that the sequence $(C_i)_{i\in\mathbb{N}}$ where $C_{i+1}=C_i \sqcup \varGamma_{G,m,t}(C_i)$ satisfies $C_i(q)=\{x\in\mathbb{R} \mid 0\leq x\leq r^i-1\}$.

Recall that a Kleene iteration algorithm applied on the computation of $\varGamma_{G,m}(C_0)$ consists in computing the beginning of the sequence $(C_i)_{i\in\mathbb{N}}$ defined by the induction $C_{i+1}=C_i \bigsqcup_{t\in T}\varGamma_{G,m,t}(C_i)$ until an integer i such that $C_{i+1}=C_i$ is discovered. Then the algorithm terminates and it returns C_i. In fact, in this case we have $C_i=\varGamma_{G,m}(C_0)$. However, as proved by the previous Example 5.1 the Kleene iteration does not terminate in general. Nevertheless we are going to compute $\varGamma_{G,m}(C_0)$ by a Kleene iteration such that each C_i is *safely* enlarged into a C_i' satisfying $C_i \sqsubseteq C_i' \sqsubseteq \varGamma_{G,m}(C_0)$. This enlargement follows the acceleration framework introduced in [LS07b, LS07a] that roughly consists to compute the precise effect of iterating some cycles. This framework motivate the introduction of the monotonic function $\varGamma_{G,m}^W$ defined over the complete lattice $(Q\rightarrow\mathcal{C}_m,\sqsubseteq)$ for any $C\in Q\rightarrow\mathcal{C}_m$ and for any $q\in Q$ by the following equality:

$$\varGamma_{G,m}^W(C)(q)=\bigsqcup_{q\xrightarrow{\sigma}q}\varGamma_{r,m,\sigma}(C(q))$$

The following Proposition 5.2 shows that $\varGamma_{G,m}^W(C)$ is effectively computable from C and the function $\varLambda_{G,m}$ introduced in section 3. In this proposition, G_q denotes the graph G reduced to the strongly connected components of q.

Proposition 5.2. *For any $C\in Q\rightarrow\mathcal{C}_m$, and for any $q\in Q$, we have:*

$$\varGamma_{G,m}^W(C)(q)=C(q)+\mathbb{R}_+(C(q)-\varLambda_{G_q,m}(q))$$

We now prove that the enlargement is sufficient to enforce the convergence of a Kleene iteration.

Table 1. The values of $C_{I,0}$ and $C_I=\varGamma_{G_I,2}(C_{I,0})$

q	$C_{I,0}(q)$	$\varGamma_{G_I,2}(C_{I,0})(q)$
1	$\{(0,0)\}$	$\mathbb{R}_+(1,3)$
2	$\emptyset$	$(1,1)+\mathbb{R}_+(3,2)$
3	$\emptyset$	$\mathbb{R}_+(3,2)$
4	$\emptyset$	$(1,0)+\mathbb{R}_+(3,2)$
5	$\emptyset$	$(0,1)+\mathbb{R}_+(1,3)$
6	$\emptyset$	$(2,1)+\mathbb{R}_+(3,2)$
7	$\emptyset$	$(0,2)+\mathbb{R}_+(1,3)$
8	$\emptyset$	$\mathrm{conv}(\{(1,0),(1,2)\})+\mathbb{R}_+(1,0)+\mathbb{R}_+(1,3)$
9	$\emptyset$	$(0,1)+\mathbb{R}_+(0,1)+\mathbb{R}_+(3,2)$

Proposition 5.3. *Let $C_0 \sqsubseteq C'_0 \sqsubseteq C_1 \sqsubseteq C'_1 \sqsubseteq \ldots$ be the sequence defined by the induction $C_{i+1} = C'_i \bigsqcup_{t\in T} \Gamma_{G,m,t}(C'_i)$ and $C'_i = \Gamma^W_{G,m}(C_i)$. There exists $i < |Q|$ satisfying $C_{i+1} = C_i$. Moreover, for such an integer i we have $C_i = \Gamma_{G,m}(C_0)$.*

Proof. Observe that $C_i \sqsubseteq C'_i \sqsubseteq \Gamma_{G,m}(C_0)$ for any $i \in \mathbb{N}$. Thus, if there exists $i \in \mathbb{N}$ such that $C_{i+1} = C_i$ we deduce that $C_i = \Gamma_{G,m}(C_0)$. Finally, in order to get the equality $C_{|Q|} = C_{|Q|-1}$, just observe by induction over i that we have following equality for any $q_2 \in Q$:

$$C'_i(q_2) = \bigsqcup_{\substack{q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma} q_1 \xrightarrow{\sigma_2} q_2 \\ |\sigma_1|+|\sigma_2| \leq i}} \Gamma_{G,m,\sigma_1\sigma\sigma_2}(C_0(q_1))$$

□

Example 5.4. Let us consider the 2-graph G_I obtained from the 2-graph depicted in the center of Fig. 1 and restricted to the set of states $Q_I = \{1,\ldots,9\}$. Let us also consider the function $C_{I,0} \in Q_I \to \mathcal{C}_2$ defined by $C_{I,0}(1) = \{(0,0)\}$ and $C_{I,0}(q) = \emptyset$ for $q \in \{2,\ldots,9\}$. Computing inductively the sequence $C_0 \sqsubseteq C'_0 \sqsubseteq C_1 \sqsubseteq C'_1 \sqsubseteq \ldots$ defined in Proposition 5.3 from $C_0 = C_{I,0}$ shows that $C_6 = C_5$. Moreover, this computation provides the value of $C_I = \Gamma_{G_I,2}(C_{I,0})$ (see Table 1).

```
FixPoint($G = (Q, \Sigma_r, T)$ a $m$−graph, $m \in \mathbb{N}\backslash\{0\}$, $C_0 \in Q \to \mathcal{C}_m$)
    let  $C \leftarrow C_0$
    while there exists  $t \in T$ such that $\Gamma_{G,m,t}(C) \not\sqsubseteq C$
        $C \leftarrow \Gamma^W_{G,m}(C)$
        let  $C \leftarrow C \sqcup \bigsqcup_{t\in T} \Gamma_{G,m,t}(C)$
    return $C$
```

Corollary 5.5. *The algorithm FixPoint(G,m,C_0) terminates by iterating the main while loop at most $|Q|-1$ times. Moreover, the algorithm returns $\Gamma_{G,m}(C_0)$.*

From Propositions 3.4 and 3.5 and corollaries 4.4 and 5.5 we get:

Theorem 5.6. *The closed convex hull of sets symbolically represented by arithmetic automata are rational polyhedral and computable in exponential time.*

Example 5.7. We follow notations introduced in Examples 3.1, 3.2 and 5.4. Observe that $C_I(8) - C_D(a) = \text{conv}(\{(1,0),(1,2)\}) + \mathbb{R}_+(1,0) + \mathbb{R}_+(1,3)$ is exactly the closed convex hull of $X = \{x \in \mathbb{N}^2 \mid 3x[1] > x[2]\}$.

6 Conclusion

We have proved that the closed convex hull of sets symbolically represented by arithmetic automata are rational polyhedral. Our approach is based on acceleration in convex data-flow analysis. It provides a simple algorithm for computing this set. Compare to [Lat04] (1) our algorithm has the same worst case exponential time complexity, (2) it is not limited to sets of the form $\mathbb{Z}^m \cap C$ where C is a rational polyhedral convex set, (3) it can be applied to any set definable in $\text{FO}(\mathbb{R}, \mathbb{Z}, +, \leq, X_r)$, (4) it can be easily implemented, and (5) it is not restricted to the most significant digit first decomposition. This last advantage directly comes from the class of arithmetic automata we consider. In fact, since the arithmetic automata can be non deterministic, our algorithm can be applied to least significant digit first arithmetic automata just by flipping the direction of the transitions. Finally, from a practical point of view, as the arithmetic automata representing sets in the restricted logic $\text{FO}(\mathbb{R}, \mathbb{Z}, +, \leq)$ (where X_r is discarded) have a very particular structure, we are confident that the exponential time complexity algorithm can be applied on automata with many states like the one presented in [Lat04]. The algorithm will be implemented in TaPAS [LP08] (The Talence Presburger Arithmetic Suite) as soon as possible.

References

[BDEK07] Becker, B., Dax, C., Eisinger, J., Klaedtke, F.: Lira: Handling constraints of linear arithmetics over the integers and the reals. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 307–310. Springer, Heidelberg (2007)

[BH06] Boigelot, B., Herbreteau, F.: The power of hybrid acceleration. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 438–451. Springer, Heidelberg (2006)

[BJW05] Boigelot, B., Jodogne, S., Wolper, P.: An effective decision procedure for linear arithmetic over the integers and reals. ACM Trans. Comput. Log. 6(3), 614–633 (2005)

[BLP06] Bardin, S., Leroux, J., Point, G.: Fast extended release. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 63–66. Springer, Heidelberg (2006)

[BRW98] Boigelot, B., Rassart, S., Wolper, P.: On the expressiveness of real and integer arithmetic automata (extended abstract). In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 152–163. Springer, Heidelberg (1998)

[FL05] Finkel, A., Leroux, J.: The convex hull of a regular set of integer vectors is polyhedral and effectively computable. Information Processing Letter 96(1), 30–35 (2005)

[Kar76] Karr, M.: Affine relationships among variables of a program. Acta Informatica 6, 133–151 (1976)

[Lat04] Latour, L.: From automata to formulas: Convex integer polyhedra. In: 19th IEEE Symposium on Logic in Computer Science (LICS 2004), Turku, Finland, July 14-17, 2004, pp. 120–129. IEEE Computer Society, Los Alamitos (2004)

[Ler04] Leroux, J.: The affine hull of a binary automaton is computable in polynomial time. In: Verification of Infinite State Systems, 5th International Workshop, INFINITY 2003, Marseille, France, September 2, 2003, vol. 98, pp. 89–104. Elsevier, Amsterdam (2003)

[Ler05] Leroux, J.: A polynomial time presburger criterion and synthesis for number decision diagrams. In: 20th IEEE Symposium on Logic in Computer Science (LICS 2005), Chicago, IL, USA, June 26-29, 2005, pp. 147–156. IEEE Computer Society, Los Alamitos (2005)

[LP08] Leroux, J., Point, G.: TaPAS: The Talence Presburger Arithmetic Suite (submited, 2008)

[LS07a] Leroux, J., Sutre, G.: Accelerated data-flow analysis. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, Springer, Heidelberg (2007)

[LS07b] Leroux, J., Sutre, G.: Acceleration in convex data-flow analysis. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 520–531. Springer, Heidelberg (2007)

[Lug04] Lugiez, D.: From automata to semilinear sets: A logical solution for sets L(C, P). In: Domaratzki, M., Okhotin, A., Salomaa, K., Yu, S. (eds.) CIAA 2004. D. Lugiez, vol. 3317, pp. 321–322. Springer, Heidelberg (2005)

[Sch87] Schrijver, A.: Theory of Linear and Integer Programming. John Wiley and Sons, New York (1987)

Pointer Analysis, Conditional Soundness, and Proving the Absence of Errors

Christopher L. Conway[1], Dennis Dams[2], Kedar S. Namjoshi[2], and Clark Barrett[1]

[1] New York University, Dept. of Computer Science
{cconway,barrett}@cs.nyu.edu
[2] Bell Laboratories, Alcatel-Lucent
{dennis,kedar}@research.bell-labs.com

Abstract. It is well known that the use of points-to information can substantially improve the accuracy of a static program analysis. Commonly used algorithms for computing points-to information are known to be sound only for memory-safe programs. Thus, it appears problematic to utilize points-to information to verify the memory safety property without giving up soundness. We show that a sound combination is possible, even if the points-to information is computed separately and only conditionally sound. This result is based on a refined statement of the soundness conditions of points-to analyses and a general mechanism for composing conditionally sound analyses.

1 Introduction

It is well known that information about pointer relationships is essential for effective analysis and optimization of C programs [2,18]. Such information can be provided by a variety of algorithms that compute an approximation of the *points-to* relations of a program (e.g., [3,12,27]). For variables `x` and `y`, `x` *points to* `y` if there is some execution of the program such that the value of `x` is either the address of `y` or, if `x` and `y` are aggregate objects (such as arrays or structures), the value of an element of `x` is an address within the extent of `y`.

A (may) points-to analysis is sound if the relation it computes over-approximates the true points-to relation of the program. Typical analysis algorithms are known to be sound only for "well-behaved" programs, i.e., programs with behavior that is well-defined by the C standard [22]. For instance, typical points-to analysis algorithms consider the points-to sets of pointer values `x` and `x+1` to be the same. This is justified if `x+1` does not "overflow" the bounds of the object pointed to by `x`. However, if the expression does overflow (i.e., the program is not "well-behaved"), the object pointed to by `x+1` is undefined.

In extending the Orion static analyzer [11] to verify memory safety, we found that performing the analysis without access to points-to information resulted in an overwhelming number of false alarms. In principle, a single, combined analysis can be defined that computes memory safety and points-to information simultaneously. Since the memory safety information being computed in one

© Springer-Verlag Berlin Heidelberg 2008

"half" is available to the points-to "half," the points-to information is kept sound even for ill-behaved executions. Conversely, the memory safety component has access to up-to-date points-to information, enabling a higher precision analysis.

However, such a fine-grained combination of analyses may not be scalable. We would like to treat existing scalable points-to analyses as plug-in components. Moreover, one may wish to perform the memory safety analysis in a "bottom-up" fashion, computing a general summary for each function on any possible input—in this case, separately computed points-to information helps limit the possible values of pointer parameters and global variables.

These considerations lead to the central questions addressed in this paper: Is it possible to obtain a sound combination of independent points-to and memory safety analyses, especially as the first obtains sound results assuming memory safety? More generally, under what conditions can conditionally sound analyses be combined? What guarantees can be made for the combination?

This paper makes several contributions:

– We formalize the notion of conditional soundness, show how to compose conditionally sound analyses, and derive the conditional soundness guarantee of the composition. Although we describe conditional soundness in the specific context of points-to analysis and a particular kind of memory safety, we believe our framework can be used to refine the soundness results of a variety of static analyses, e.g., analyses that are sound assuming sequential consistency or numerical analyses that are sound assuming the absence of integer overflows. Conditional soundness can be formulated in terms of the (unconditional) preservation of a class of temporal safety properties using Cousot and Cousot's power construction [7,8,9]. Our formulation, while more specialized, is simpler (e.g., it is state-based rather than path-based) and captures the behavior of several interesting analyses more directly.
– We show that a set of points-to analyses similar to and sharing the soundness properties of commonly-used flow-sensitive and insensitive analyses—such as those of Emami, Ghiya, and Hendren [16]; Wilson and Lam [28]; Andersen [3]; Steensgaard [27]; and Das [12]—provide results that are sound for any *memory-safe execution* of a program. This statement is both stronger and more precise than the traditional statement that such analyses are sound for "well-behaved" programs.
– This more precise characterization of a points-to analysis, along with the combination theorem for conditional analyses, shows that the combination of an independent points-to analysis with a memory safety analysis is conditionally sound. The soundness result guarantees that the *absence* of errors can be proved. Conversely, for a program with memory errors, at least one representative error—but not necessarily all errors—along any unsafe execution will be detected.

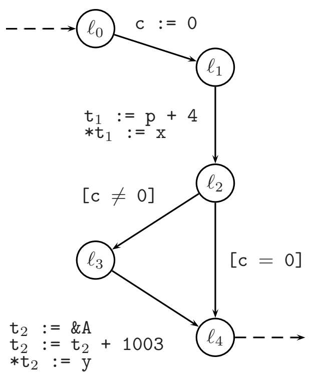
Motivating Example. Figure 1(a) is an example of a program which is *not* well-behaved: there is an off-by-one error at label `L1` and an off-by-one-thousand error at label `L3` when `c` is not `0`. Assume that the functions `ok` and `bad` are analyzed in a bottom-up fashion, without reference to the actual parameters

```
int A[4], c;

void bad(int *p, int x, int y) {
L0: c = 0;
L1: p[4] = x;
L2: if( c!=0 ) {
L3:     A[1003] = y;
L4: }
}

void ok(int *q, int n) {
L5: q[0] = n;
}

void main() {
    ok(A,0);
    bad(A,1,0);
}
```

(a)

$$\mathcal{V} = \{\mathtt{A}, \mathtt{c}, \mathtt{p}, \mathtt{x}, \mathtt{y}, \mathtt{t}_1, \mathtt{t}_2\}$$

$$\Gamma(v) = \begin{cases} 4, & \text{if } v = \mathtt{A} \\ 1, & \text{otherwise} \end{cases}$$



(b)

Fig. 1. An unsafe C program

supplied in `main`. Without any points-to information regarding `q`, the only safe assumption is that the expression `q[0]` at label `L5` can alias any location in memory—a conservative memory safety analysis would be forced to assume that the behavior of the program is undefined from this point on. But this is not the case: the function `ok` is memory-safe so long as `q` points to an array of at least one element. Points-to information is necessary to obtain a precise memory safety analysis.

A typical points-to analysis (e.g., Andersen's [3]) will determine that `p` and `q` both point to `A` and not to `c`, `n`, `x`, or `y`. Using points-to information, a memory safety analysis can (correctly) infer that `q[0]` is an in-bounds location at `L5` and, thus, the execution of `ok` is well-defined. Further, it can (correctly) detect that `p[4]` is out-of-bounds at `L1` and emit a useful error report.

In many implementations `p[4]` will alias `c` at `L1`—so that `c` is set to `1`, making `L3` reachable—but `p` will not point to `c` according to the points-to relation. Since `c` is initialized to `0` in `bad` and—according to the points-to relation—no expression aliasing `c` is subsequently assigned to, the analysis is likely to (incorrectly) infer that the error at `L3` is unreachable. Thus, it may seem that relying on the points-to relation will lead a static analyzer to miss real errors. Note, though, that the reachability of the error at `L3` is due solely to the unsafe assignment at `L1`: the points-to relation can be relied upon up to the first occurrence of a memory safety error.

This line of reasoning is not specific to the example: as we will show, it applies to any conditionally sound analysis, enabling such an analysis to detect at least one error along any erroneous execution.

Note: Full proofs of all theorems in this paper are given in a technical report [5].

2 Program Analysis and Conditional Soundness

To present program analysis in a formal setting, we use the framework of abstract interpretation [6]. A full syntax of program statements is given in the next section. For now, we are concerned only with the relationship between concrete and abstract interpretations. We omit any discussion of techniques (such as widening and extrapolation) which serve to make program analyses finite and computable—we are concerned solely with issues of soundness.

Let C be a distinguished set of *concrete states*. A *domain* (D, γ) is a pair, where D is a set of *abstract states* and $\gamma : D \to 2^C$ is a *concretization function*. When the meaning is clear, we overload D to refer both to a domain and to its underlying set of states and use γ_D to refer to the concretization function. We lift γ_D to sets of states: $\gamma_D(D') = \bigcup_{d \in D'} \gamma_D(d)$, where $D' \subseteq D$. We say a set $D' \subseteq D$ *over-approximates* $C' \subseteq C$ iff $\gamma_D(D') \supseteq C'$.

We define the soundness of a program interpretation in terms of a *collecting semantics*. Given a (concrete or abstract) domain D, we will define a *semantic operator* $[\![\cdot]\!]$ which maps a program $\mathcal{P}$ to a set $[\![\mathcal{P}]\!] \subseteq D$ of *reachable states*. The semantics $[\![\mathcal{P}]\!]$ is defined inductively in terms of *semantic interpretations over* D: a set $\mathcal{I}[\mathcal{P}] \subseteq D$ of *initial states* and a *transfer function* $\mathcal{F}[\mathcal{P}] : D \to 2^D$. We lift $\mathcal{F}[\mathcal{P}]$ to sets of states: $\mathcal{F}[\mathcal{P}](D') = \bigcup_{d \in D'} \mathcal{F}[\mathcal{P}](d)$, where $D' \subseteq D$.

An *analysis* $\mathcal{A}$ is represented as a tuple $(D, \mathcal{I}, \mathcal{F})$, where D is a domain and $\mathcal{I}$ and $\mathcal{F}$ are semantic interpretations over D. We use $D_{\mathcal{A}}$, $\mathcal{I}_{\mathcal{A}}$, and $\mathcal{F}_{\mathcal{A}}$ to denote the constituents of an analysis $\mathcal{A}$ and $\gamma_{\mathcal{A}}$ to denote the concretization function of the domain $D_{\mathcal{A}}$.

Definition 1. *Let $\mathcal{A} = (D, \mathcal{I}, \mathcal{F})$ be an analysis. The k-*reachability predicate* $R^k_{\mathcal{A}}[\mathcal{P}]$ *for a program* $\mathcal{P}$ w.r.t. $\mathcal{A}$ *holds if a state is reachable in $\mathcal{A}$ in exactly k steps. We define $R^k_{\mathcal{A}}[\mathcal{P}]$ inductively as a subset of D:*

$$R^0_{\mathcal{A}}[\mathcal{P}] = \mathcal{I}[\mathcal{P}] \qquad\qquad R^k_{\mathcal{A}}[\mathcal{P}] = \mathcal{F}[\mathcal{P}](R^{k-1}_{\mathcal{A}}[\mathcal{P}]), \quad k > 0$$

The semantics $[\![\cdot]\!]_{\mathcal{A}}$ w.r.t. $\mathcal{A}$ *maps a program $\mathcal{P}$ to a subset of D, the* reachable states in $\mathcal{P}$ w.r.t. $\mathcal{A}$*:*

$$[\![\mathcal{P}]\!]_{\mathcal{A}} = \bigcup_{k \geq 0} R^k_{\mathcal{A}}[\mathcal{P}]$$

To judge the soundness of an analysis, we need a concrete semantics against which it can be compared. The *concrete domain* $D_{\mathcal{C}}$ is given by the pair (C, γ_C), where γ_C is the trivial concretization function: $\gamma_C(c) = \{c\}$. We assume that a *concrete analysis* $\mathcal{C} = (D_{\mathcal{C}}, \mathcal{I}_{\mathcal{C}}, \mathcal{F}_{\mathcal{C}})$ is given. The concrete analysis uniquely defines a *concrete semantics* $[\![\cdot]\!]_{\mathcal{C}}$.

Definition 2. *An analysis $\mathcal{A}$ is* sound *iff for every program $\mathcal{P}$, $[\![\mathcal{P}]\!]_{\mathcal{A}}$ over-approximates $[\![\mathcal{P}]\!]_{\mathcal{C}}$ (i.e., $\gamma_{\mathcal{A}}([\![\mathcal{P}]\!]_{\mathcal{A}}) \supseteq [\![\mathcal{P}]\!]_{\mathcal{C}}$).*

Conditional Soundness. So far, we have defined a style of analysis which is *unconditionally sound*, mirroring the traditional approach to abstract interpretation. However, as we have noted, points-to analysis is sound only under certain assumptions about the behavior of the program analyzed. To address this, we introduce the notion of *conditional soundness with respect to a predicate θ*. An analysis will be *θ-sound* if it over-approximates the concrete states of a program that are reachable via *only θ-states*. We first define a semantics restricted to θ.

Definition 3. *Let $\mathcal{A} = (D, \mathcal{I}, \mathcal{F})$ be an analysis and θ a predicate on D (we view the predicate θ, equivalently, as a subset of D). The* θ-restricted k-reachability predicate $R^k_{\mathcal{A}}{\downarrow_\theta}[\mathcal{P}]$ for program $\mathcal{P}$ w.r.t. $\mathcal{A}$ *holds if a state is reachable in $\mathcal{A}$ in exactly k steps via* only θ states. *$R^k_{\mathcal{A}}{\downarrow_\theta}[\mathcal{P}]$ is defined inductively:*

$$R^0_{\mathcal{A}}{\downarrow_\theta}[\mathcal{P}] = \mathcal{I}[\mathcal{P}] \qquad R^k_{\mathcal{A}}{\downarrow_\theta}[\mathcal{P}] = \mathcal{F}[\mathcal{P}](\theta \cap R^{k-1}_{\mathcal{A}}{\downarrow_\theta}[\mathcal{P}]), \quad k > 0$$

The θ-restricted semantics $[\![\cdot]\!]_{\mathcal{A}}{\downarrow_\theta}$ w.r.t. $\mathcal{A}$ *maps a program $\mathcal{P}$ to a subset of D, the* θ-reachable states in $\mathcal{P}$ w.r.t. $\mathcal{A}$*:*

$$[\![\mathcal{P}]\!]_{\mathcal{A}}{\downarrow_\theta} = \bigcup_{k \geq 0} R^k_{\mathcal{A}}{\downarrow_\theta}[\mathcal{P}]$$

Note that $R^k_{\mathcal{A}}{\downarrow_\theta}[\mathcal{P}]$ may include non-θ states—neither $\mathcal{I}[\mathcal{P}]$ nor the range of $\mathcal{F}[\mathcal{P}]$ are restricted to θ—but those states will not yield successors in $R^{k+1}_{\mathcal{A}}{\downarrow_\theta}[\mathcal{P}]$. The θ-restricted semantics give us a lower bound for the approximation of a θ-sound analysis.

Definition 4. *Let $\mathcal{A}$ be an analysis and θ a predicate on C. $\mathcal{A}$ is* θ-sound *iff for every program $\mathcal{P}$, $[\![\mathcal{P}]\!]_{\mathcal{A}}$ over-approximates $[\![\mathcal{P}]\!]_{\mathcal{C}}{\downarrow_\theta}$.*

Note that an unconditionally sound analysis is also θ-sound for *any* θ. More generally, any θ-sound analysis is also φ-sound, for any φ stronger than θ.

This notion of conditional soundness does not just give us a more precise statement of the behavior of certain analyses—it provides us with a sufficient condition to show an analysis *proves the absence of error states.*

Theorem 1. *Let $\mathcal{P}$ be a program and $\mathcal{A}$ a θ-sound analysis. If there are no reachable non-θ states in $\mathcal{P}$ w.r.t. $\mathcal{A}$, then there are no reachable concrete non-θ states in $\mathcal{P}$ (i.e., if $\gamma_A([\![\mathcal{P}]\!]_{\mathcal{A}}) \subseteq \theta$, then $[\![\mathcal{P}]\!]_{\mathcal{C}} \subseteq \theta$)).*

In Section 4, we will show that points-to analysis is SafeDeref-sound, where SafeDeref is a predicate that captures memory safety.

Parameterized Analysis. Having defined a precise notion of conditional soundness, we now consider how the results of a θ-sound analysis can be used to refine a second analysis. Suppose that $\mathcal{A}$ is an analysis and we have already computed the set of reachable states $[\![\mathcal{P}]\!]_{\mathcal{A}}$. We may wish to use the information present in $[\![\mathcal{P}]\!]_{\mathcal{A}}$ to refine a second analysis over a different domain B. For example, we could use the reduced product construction [7] to form a new domain over a subset of $D_{\mathcal{A}} \times B$ including only those states (a, b) where a is in $[\![\mathcal{P}]\!]_{\mathcal{A}}$ and the states a and b "agree" (e.g., $\gamma_{\mathcal{A}}(a) \cap \gamma_B(d) \neq \emptyset$).

Traditional methods for combining analyses take a "white box" approach—e.g., Cousot and Cousot [7] assume that the state transformers are available and can be combined in a mechanical way; Lerner et al. [24] assume that analyses can be run in parallel, one step at a time. In contrast, we will assume that any prior analysis is a black box: we have access to its result (in the form of a set of reachable abstract states), its domain (which allows us to interpret the result), and some (possibly conditional) soundness guarantee. This naturally models the use of off-the-shelf program analyses to provide refinement advice.

We will define such a refinement in terms of a parameterized analysis which produces a new, refined analysis from the results of a prior analysis. An *analysis generator* $\widetilde{\mathcal{G}}$ is a tuple $(D, E, \widetilde{\mathcal{I}}, \widetilde{\mathcal{F}})$ where: D and E are domains (the *input* and *output* domains, respectively) and $\widetilde{\mathcal{I}}$ and $\widetilde{\mathcal{F}}$ are *parameterized interpretations* mapping a set of states $D' \subseteq D$ to semantic interpretations $\widetilde{\mathcal{I}}\langle D' \rangle$ and $\widetilde{\mathcal{F}}\langle D' \rangle$ over E. We denote by $\widetilde{\mathcal{G}}\langle D' \rangle$ the analysis over E defined by the parameterized interpretations on input D': $\widetilde{\mathcal{G}}\langle D' \rangle = (E, \widetilde{\mathcal{I}}\langle D' \rangle, \widetilde{\mathcal{F}}\langle D' \rangle)$. As one might expect, the soundness of $\widetilde{\mathcal{G}}\langle D' \rangle$ depends on the input D'.

Definition 5. *An analysis generator $\widetilde{\mathcal{G}}$ with input domain D is* sound *iff for every set of states $D' \subseteq D$, $\widetilde{\mathcal{G}}\langle D' \rangle$ is θ-sound with $\theta = \gamma_D(D')$.*

Given an analysis generator, it is natural to consider the analysis formed by composing the generator with an analysis over its input domain. If $\mathcal{A}$ is an analysis and $\widetilde{\mathcal{G}}$ is an analysis generator with input domain $D_{\mathcal{A}}$ (i.e., the input domain of $\widetilde{\mathcal{G}}$ is the underlying domain of $\mathcal{A}$), the *composed analysis* $\widetilde{\mathcal{G}} \circ \mathcal{A}$ is defined by providing the result of $\mathcal{A}$ as a parameter to $\widetilde{\mathcal{G}}$ (i.e., $\widetilde{\mathcal{G}} \circ \mathcal{A} = \widetilde{\mathcal{G}}\langle [\![\mathcal{P}]\!]_{\mathcal{A}} \rangle$). An important property of the composed analysis is preservation of soundness.

Theorem 2. *If $\widetilde{\mathcal{G}}$ is sound and $\mathcal{A}$ is θ-sound, then the composed analysis $\widetilde{\mathcal{G}} \circ \mathcal{A}$ is θ-sound.*

In the remainder of this paper, we will apply Theorem 2 to the problem of verifying memory safety using points-to information. In Section 3, we define the concrete semantics for a little language that captures the pointer semantics of C. In Section 4, we define a memory safety property SafeDeref and a set of SafeDeref-sound points-to analyses similar to the points-to analyses found in the literature. In Section 5, we define a memory safety analysis parameterized by points-to information. These, together with Theorem 2, allow us to prove the absence of memory safety errors.

Note that since the points-to analysis is SafeDeref-sound, we could in theory use it to prove the absence of memory safety errors. However, the points-to domain does not track memory safety information with adequate precision to detect errors without an impractical number of false alarms. The value of Theorem 2 is that it allows for the definition of a specialized memory safety analysis which uses the points-to analysis to increase its precision while remaining SafeDeref-sound.

$$n \in \mathbb{Z} \qquad \mathtt{x}, \mathtt{y} \in \mathit{Vars}$$

$$\begin{aligned} L \in \mathit{Lvals} &::= \mathtt{x} \mid \mathtt{*x} \\ E \in \mathit{Exprs} &::= L \mid n \mid \mathtt{x} \oplus \mathtt{y} \mid \mathtt{x} \trianglelefteq \mathtt{y} \mid \mathtt{\&x} \\ S \in \mathit{Stmts} &::= L \; \mathtt{:=} \; E \mid \mathtt{[}E\mathtt{]} \end{aligned}$$

Fig. 2. Grammar for a minimal C-like language

3 Concrete Semantics

To make precise statements about program analyses requires a concrete program semantics. We will define the semantics of the little language presented in Fig. 2. The language eliminates all but those features of C that are essential to the question at hand. The semantics of the language is chosen to model the requirements of ANSI/ISO C [22] without making implementation-specific assumptions. Undefined or implementation-defined behaviors are modeled with explicit nondeterminism. Note that an ANSI/ISO-compliant C compiler is free to implement undefined behaviors in a specific, deterministic manner. By modeling undefined behaviors using non-determinism, the soundness statements made about each analysis apply to *any* standard-compliant compilation strategy.

The most important features of C that we exclude here are fixed-size integer types, narrowing casts, dynamic memory allocation, and functions.[1] We also ignore the "strict aliasing" rule [22, §6.5]. Each of these can be handled, at the cost of a higher degree of complexity in our definitions.

The syntactic classes of variables, lvalues, expressions, and statements, are defined in Fig. 2. We use n to represent an integer constant and `x` and `y` to represent arbitrary variables. We use $\oplus$ to represent an arbitrary binary arithmetic operator and $\trianglelefteq$ to represent a relational operator. Pointer operations include arithmetic, indirection (`*`), and address-of (`&`). Statements include assignments and tests ($\mathtt{[}E\mathtt{]}$, where E is an expression).

Variables in our language are viewed as arrays of memory cells. Each cell may hold either an unbounded integer or a pointer value. The only type information present is the allocated size of each variable—the "type system" merely maps variables to their sizes and provides no safety guarantees.

A *program* $\mathcal{P}$ is a tuple $(\mathcal{V}, \Gamma, \mathcal{L}, \mathcal{S}, \tau, en)$, where: $\mathcal{V} \subseteq \mathit{Vars}$ is a finite set of *program variables*; $\Gamma : \mathcal{V} \to \mathbb{N}$ is a *typing environment* mapping variables to their allocated sizes; $\mathcal{L}$ is a finite set of *program points*; $\mathcal{S} \subseteq \mathit{Stmts}$ is a finite set of *program statements* whose variables are from $\mathcal{V}$; $\tau \subseteq \mathcal{L} \times \mathcal{S} \times \mathcal{L}$ is a *transition*

[1] The omission of dynamic allocation in the discussion of points-to analysis and memory safety may seem an over-simplification. However, it is not essential to our purpose here. Points-to analyses typically handle dynamic allocation by treating each allocation site as if it were the static declaration of a global array of unknown size.

relation; and $en \in \mathcal{L}$ is a distinguished *entry point*. In the following, we assume a fixed program $\mathcal{P} = (\mathcal{V}, \Gamma, \mathcal{L}, \mathcal{S}, \tau, en)$.

Example 1. Figure 1(b) gives a fragment of the program representation for the code in Fig. 1(a), corresponding to the function `bad` . We have introduced temporaries $\mathtt{t}_1$ and $\mathtt{t}_2$ in order to simplify expression evaluation and compressed multiple statements onto a single transition when they represent a single statement in the source program.

In order to reason about points-to and memory safety analyses, we need a memory model on which to base the concrete semantics. The unit of memory allocation is a *home* in the set $\mathbb{H}$. Each home h represents a contiguous block of memory cells, e.g., a statically declared array. A *location* $h[i]$ represents the cell at integer *offset* i in home h. The set of locations with homes from $\mathbb{H}$ is denoted $\mathbb{L}$. The function $\mathbf{size} : \mathbb{H} \to \mathbb{N}$ maps a home to its allocated size. When $0 \le i < \mathbf{size}(h)$, location $h[i]$ is *in bounds*; otherwise it is *out of bounds*. Memory locations contain values from the set $\mathit{Vals} = \mathbb{Z} \cup \mathbb{L}$. A *memory state* is a partial function $m : \mathbb{L} \to \mathit{Vals}$. The set of all memory states is denoted $\mathbb{M}$. The set of concrete states C is the set of pairs (p, m) where $p \in \mathcal{L}$ represents the program position and m is a memory state.

An *allocation for* $\mathcal{V}$ is an injective function $\mathbf{home} : \mathcal{V} \to \mathbb{H}$ such that $\mathbf{size}(\mathbf{home}(\mathtt{x})) = \Gamma(\mathtt{x})$ for all $\mathtt{x} \in \mathcal{V}$. Given such an allocation, the *lvalue* of $\mathtt{x} \in \mathcal{V}$ is $\mathbf{lval}(\mathtt{x}) = \mathbf{home}(\mathtt{x})[0]$. We write $m(\mathtt{x})$ for $m(\mathbf{lval}(\mathtt{x}))$ and $m[\mathtt{x} \mapsto v]$ for $m[\mathbf{lval}(\mathtt{x}) \mapsto v]$, where m is a memory state. We say a location $h[i]$ is *within* a variable $\mathtt{x}$ if $h = \mathbf{home}(\mathtt{x})$ and $h[i]$ is in bounds.

Figure 3 defines the concrete interpretations $\mathcal{E}$ and **post** of, respectively, expressions and statements. We present here only the most interesting cases. Complete definitions are given in a technical report [5]. Note that both $\mathcal{E}$ and **post** result in *sets* of, respectively, values and concrete states—the set-based semantics is needed as undefined operations may have a nondeterministic result. $\mathcal{E}$ returns the distinguished value $\perp$ in the case where an expression is not just ill-defined, but erroneous (e.g., reading an out-of-bounds memory location)—in this case the next state can have *any* memory state at *any* program point.

We now define the concrete interpretation of a program.

Definition 6. *The* concrete semantics $[\![\mathcal{P}]\!]_{\mathcal{C}}$ *of a program* $\mathcal{P} = (\mathcal{V}, \Gamma, \mathcal{L}, \mathcal{S}, \tau, en)$ *is defined by the analysis* $\mathcal{C} = (D_{\mathcal{C}}, \mathcal{I}_{\mathcal{C}}, \mathcal{F}_{\mathcal{C}})$*, where*

$$\mathcal{I}_{\mathcal{C}}[\mathcal{P}] = \{(en, m) \mid \forall l \in \mathbb{L}.\ m(l) \text{ is not a location}\}$$

$$\mathcal{F}_{\mathcal{C}}[\mathcal{P}](p, m) = \bigcup_{(p, S, p') \in \tau} \mathbf{post}(m, p', S)$$

Example 2. Figure 4(a) gives a subset of the reachable concrete states of the program in Fig. 1(b). At ℓ_0, `p` is `A`[0] (the base address of the array `A`), `x` is 1, and `y` is 0. At ℓ_1, due to the assignment to out-of-bounds location `A`[4], the next state is undefined: *every* program point is reachable with *any* memory state.

$$\mathcal{E}(m, \mathtt{x}) = \begin{cases} \mathbb{Z}, & \text{if } m(\mathtt{x}) \text{ is undefined} \\ \{m(\mathtt{x})\}, & \text{otherwise} \end{cases}$$

$$\mathcal{E}(m, *\mathtt{x}) = \begin{cases} \bot, & \text{if } m(\mathtt{x}) \text{ is undefined, not a location, or out of bounds} \\ \mathbb{Z} & \text{if } m(m(\mathtt{x})) \text{ is undefined} \\ \{m(m(\mathtt{x}))\}, & \text{otherwise} \end{cases}$$

$$\mathbf{post}(m, p, \mathtt{x}\ \mathtt{:=}\ E) = \begin{cases} \mathcal{L} \times \mathbb{M}, & \text{if } \mathcal{E}(m, E) = \bot \\ \{(p, m[\mathtt{x} \mapsto v]) \mid v \in \mathcal{E}(m, E)\}, & \text{otherwise} \end{cases}$$

$$\mathbf{post}(m, p, *\mathtt{x}\ \mathtt{:=}\ E) = \begin{cases} \mathcal{L} \times \mathbb{M}, & \text{if } m(\mathtt{x}) \text{ is undefined, not a location, or out of bounds;} \\ & \text{or if } \mathcal{E}(m, E) = \bot \\ \{(p, m[m(\mathtt{x}) \mapsto v]) \mid v \in \mathcal{E}(m, E)\}, & \text{otherwise.} \end{cases}$$

Fig. 3. The concrete interpretation

4 Pointer Analysis

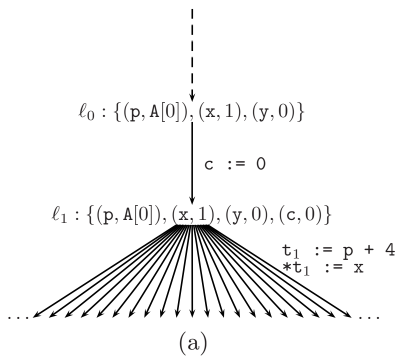
The goal of pointer analysis is to compute an over-approximate points-to set for each variable in the program, i.e., the set of homes “into” which a variable may point in some reachable state.

A *points-to state* is a relation between variables. We denote the set of points-to states by *Pts*. When it is convenient, we treat a points-to state also as a relation between variables and memory locations: for points-to state *pts*, variables $\mathtt{x}, \mathtt{y}$, and location $h[i]$, we say $(\mathtt{x}, h[i])$ is in *pts* when $(\mathtt{x}, \mathtt{y})$ is in *pts* and $h[i]$ is within $\mathtt{y}$ (i.e., $h[i]$ is in bounds and $h = \mathbf{home}(\mathtt{y})$). We write $pts(\mathtt{x})$ for the *points-to set* of the variable $\mathtt{x}$ in *pts*, i.e., the set of variables $\mathtt{y}$ (alt. locations l) such that $(\mathtt{x}, \mathtt{y})$ (alt. $(\mathtt{x}, l)$) is in *pts*.

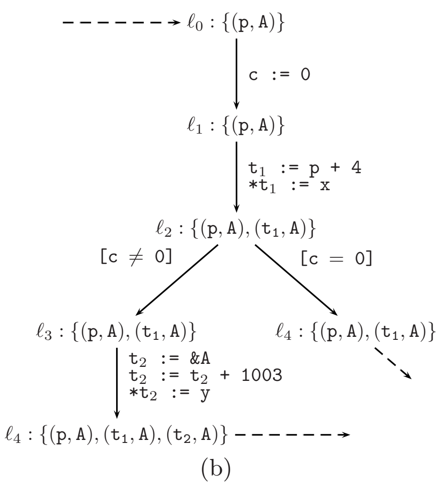
The concretization function γ_{Pts} takes a points-to state to the set of concrete states where at *most* its points-to relationships hold. Say that variable $\mathtt{x}$ *points to* $\mathtt{y}$ in memory state m if there exist locations l_1, l_2 such that l_1 is within $\mathtt{x}$, l_2 is within $\mathtt{y}$, and $m(l_1) = l_2$. Then m is in $\gamma_{Pts}(pts)$ iff for all $\mathtt{x}, \mathtt{y}$ such that $\mathtt{x}$ points to $\mathtt{y}$ in m, the pair $(\mathtt{x}, \mathtt{y})$ is in *pts*. Note that there may be other pairs in *pts* as well—the points-to relation is over-approximate. Note also that *only in-bounds location values must agree with the points-to state*; out-of-bounds locations are unconstrained.

Figure 5 defines the interpretations $\mathcal{E}_{Pts}$ and $\mathbf{post}_{Pts}$ for a selection of, respectively, expressions and statements in the points-to domain. (Complete definitions are given in a technical report [5].) The interpretations are chosen to match those used by common points-to analyses. A key feature is the treatment of the indirection operator $*$, which assumes that its argument is within bounds. Without this assumption, the interpretation would have to use the “top” points-to state (i.e., all pairs of variables) for the result of any indirect assignment.

We lift *Pts* to the set $\mathcal{L} \times Pts$ in the natural way.



$\ell_0 : \{(\mathtt{p}, \mathtt{A}[0]), (\mathtt{x}, 1), (\mathtt{y}, 0)\}$
c := 0
$\ell_1 : \{(\mathtt{p}, \mathtt{A}[0]), (\mathtt{x}, 1), (\mathtt{y}, 0), (\mathtt{c}, 0)\}$
t1 := p + 4
*t1 := x
...
...
(a)



$\ell_0 : \{(\mathtt{p}, \mathtt{A})\}$
c := 0
$\ell_1 : \{(\mathtt{p}, \mathtt{A})\}$
t1 := p + 4
*t1 := x
$\ell_2 : \{(\mathtt{p}, \mathtt{A}), (\mathtt{t_1}, \mathtt{A})\}$
[c ≠ 0]
[c = 0]
$\ell_3 : \{(\mathtt{p}, \mathtt{A}), (\mathtt{t_1}, \mathtt{A})\}$
$\ell_4 : \{(\mathtt{p}, \mathtt{A}), (\mathtt{t_1}, \mathtt{A})\}$
t2 := &A
t2 := t2 + 1003
*t2 := y
$\ell_4 : \{(\mathtt{p}, \mathtt{A}), (\mathtt{t_1}, \mathtt{A}), (\mathtt{t_2}, \mathtt{A})\}$
(b)

Fig. 4. Concrete and points-to semantics for the program in Fig. 1(b)

$$\begin{aligned}
\mathcal{E}_{Pts}(pts, \mathtt{x}) &= pts(\mathtt{x}) \\
\mathcal{E}_{Pts}(pts, *\mathtt{x}) &= \{\mathtt{z} \in \mathcal{V} \mid \exists \mathtt{y} \in \mathcal{V} : pts(\mathtt{x}, \mathtt{y}) \wedge pts(\mathtt{y}, \mathtt{z})\} \\
\mathbf{post}_{Pts}(pts, \mathtt{x} \mathrel{:=} E) &= pts \cup \{(\mathtt{x}, \mathtt{y}) \mid \mathtt{y} \in \mathcal{E}_{Pts}(pts, E)\} \\
\mathbf{post}_{Pts}(pts, *\mathtt{x} \mathrel{:=} E) &= \bigcup_{(\mathtt{x},\mathtt{y}) \in pts} \mathbf{post}_{Pts}(pts, \mathtt{y} \mathrel{:=} E)
\end{aligned}$$

Fig. 5. Abstract interpretation over points-to states

Definition 7. *A* flow- and path-sensitive points-to analysis **Pts** *is given by the tuple* $(Pts, \mathcal{I}_{Pts}, \mathcal{F}_{Pts})$, *where*

$$\begin{aligned}
\mathcal{I}_{Pts}[\mathcal{P}] &= \{(en, \emptyset)\} \\
\mathcal{F}_{Pts}[\mathcal{P}](p, pts) &= \bigcup_{(p,S,p') \in \tau} (p', \mathbf{post}_{Pts}(pts, S))
\end{aligned}$$

Example 3. Figure 4(b) shows a subset of the reachable points-to states for the program in Fig. 1(b). At ℓ_0, `p` points to `A`. The transition from ℓ_1 to ℓ_2 causes $\mathtt{t}_1$ to point to `A` as well. The presence of an out-of-bounds array access has no effect on the points-to state: the analysis assumes that evaluating $*\mathtt{t}_1$ is safe.

Definition 8. *Let* SafeDeref *be the predicate that holds in a concrete state* (p, m) *if, for every transition* (p, S, p') *in* τ *where* S *includes an expression of the form* $*\mathtt{x}$, $m(\mathtt{x})$ *is an in-bounds location.*

Theorem 3. *The points-to analysis* **Pts** *is* SAFEDEREF*-sound.*

We can extract more traditional flow-sensitive, global, and flow-insensitive pointer analyses from $[\![\mathcal{P}]\!]_{\mathbf{Pts}}$ as follows.

– A *flow-sensitive, program-point-sensitive (path-insensitive) analysis* is derived by assigning to each program point p the least points-to state (by subset inclusion) $pts^\sharp$ such that, if (p, pts) is in $[\![\mathcal{P}]\!]_{\mathbf{Pts}}$, then $pts \subseteq pts^\sharp$.
– A *flow-sensitive, global (program-point-insensitive) analysis* is derived by assigning to every program point the least points-to state (by subset inclusion) $pts^\sharp$ such that, if (p, pts) is in $[\![\mathcal{P}]\!]_{\mathbf{Pts}}$ for *any* program point p, then $pts \subseteq pts^\sharp$.
– A *flow-insensitive analysis* is derived by replacing τ in Definition 7 with the relation $\tau^\sharp$, where the edge (p, S, q) is in $\tau^\sharp$ whenever some edge (t, S, u) is in τ, for *any* program points t and u. Intuitively, if a statement occurs anywhere in the program, then it may occur between any two program points—the interpretation ignores the control-flow structure of the program.
– *Flow-insensitive, program-point-sensitive* and *flow-insensitive, global* combinations can be defined as above, substituting the flow-insensitive semantics for $[\![\mathcal{P}]\!]_{\mathbf{Pts}}$.

Theorem 4. *Each of the flow-, path-, and program-point-sensitive and insensitive variations of the points-to analysis is* SAFEDEREF*-sound.*

Note 1. The flow-sensitive, program-point-sensitive analysis yields a points-to relation similar to that of Emami et al. [16]. The flow-insensitive, global analysis procedure yields a points-to relation similar to that of Andersen [3]. The Steensgaard [27] and Das [12] relations add additional approximation to the global relation. We claim (but do not prove formally here) that these procedures approximate $[\![\mathcal{P}]\!]_{\mathbf{Pts}}$ and, thus, are *at least* SAFEDEREF-sound.

By the definition of conditional soundness, it is possible some condition θ weaker than SAFEDEREF exists such that some or all of the above analyses are θ-sound. It is our belief that this is not the case: no realistic points-to analysis is θ-sound for any θ weaker than SAFEDEREF. A proof of this proposition is beyond the scope of this paper.

In summary, we have shown that a set of points-to analyses which share the assumptions of widely used analyses from the literature are sound for all memory-safe executions. This claim is both stronger and more precise than any correctness claims the authors have encountered: our points-to analyses (and, by extension, those cited above) compute a relation which is conservative not only for "well-behaved" (i.e., memory-safe) programs, but for all well-behaved *executions*, even the well-behaved executions of ill-behaved programs

We have shown that, if we can prove the absence of non-SAFEDEREF states in $[\![\mathcal{P}]\!]_{\mathcal{C}}$, the points-to analyses we have defined above will be sound. It remains to describe an analysis parameterized by points-to information which can perform a precise memory safety analysis.

5 Checking Memory Safety

We wish to define an analysis procedure that will soundly prove the absence of non-SafeDeref states in the concrete program. Note that the only attributes of a location value that are relevant to the property SafeDeref are its offset and the size of its home; if we can precisely track these attributes, we can ignore the home component of a location (i.e., which variable it is within) so long as we have access to over-approximate points-to information.

Note 2. In our description of the analysis, we will omit the merging, widening, and covering operations necessary to make the reachability computation tractable. In our implementation of a memory safety analysis in Orion, we constrain integer values and pointer offsets using a relational abstract domain (e.g., convex polyhedra [10]) and use merging and widening to efficiently over-approximate the semantics given below.

Our analysis will track *abstract values* from the set $\widehat{Vals}$. An abstract value is either an integer or an *abstract location*, a pair (i, n) representing a location at offset i in a home of size n. Each abstract value $\hat{v}$ represents a set of concrete values, according to the abstraction function $\alpha : Vals \rightarrow \widehat{Vals}$. For integer values, α is the identity (i.e., $\alpha(n) = n$); for concrete location values, α preserves the offset and size (i.e., $\alpha(h[i]) = (i, \mathbf{size}(h))$). An abstract location (i, n) is *in bounds* if it represents only in bounds concrete locations (i.e., $0 \leq i < n$); otherwise it is *out of bounds*. An *abstract memory state* is a partial function $b : \mathbb{L} \rightarrow \widehat{Vals}$. We denote by B the set of abstract memory states.

The concretization function $\gamma_B : B \rightarrow 2^C$ takes an abstract memory state b to the set of concrete memories abstracted by b. A concrete memory m is in $\gamma_B(b)$ iff for all l either $m(l)$ and $b(l)$ are both undefined or $\alpha(m(l)) = b(l)$.

Figure 6 defines the interpretations $\mathcal{E}_B$ and $\mathbf{post}_B$ for a selection of, respectively, expressions and statements with respect to B. (Complete definitions are given in a technical report [5].) Note that the interpretations rely on points-to information. In the limiting case, where no points-to information is available (i.e., the points-to relation includes all pairs), the expression `*x` can take the value of any location abstracted by $b(\mathtt{x})$. As in the concrete interpretation $\mathcal{E}_B$ returns the value $\perp$ in the case where expression evaluation is (potentially) erroneous.

We lift B to the domain $\mathcal{L} \times B$ in the natural way.

Definition 9. *The analysis generator $\widetilde{\mathcal{B}}$ maps a set of points-to states Q to the memory safety analysis $\widetilde{\mathcal{B}}\langle Q\rangle$ defined by the parameterized interpretations*

$$\widetilde{\mathcal{I}}_{\mathcal{B}}\langle Q\rangle[\mathcal{P}] = \{(en, b) \mid \forall l \in \mathbb{L} : b(l) \text{ is undefined}\}$$

$$\widetilde{\mathcal{F}}_{\mathcal{B}}\langle Q\rangle[\mathcal{P}](p, b) = \bigcup_{(p,S,p') \in \tau} \; \bigcup_{(p,pts) \in Q} \mathbf{post}_B(b, pts, p', S)$$

Theorem 5. *The analysis generator $\widetilde{\mathcal{B}}$ is sound.*

$$\mathcal{E}_B(b, pts, \mathtt{x}) = \begin{cases} \mathbb{Z}, & \text{if } b(\mathtt{x}) \text{ is undefined} \\ \{b(\mathtt{x})\}, & \text{otherwise} \end{cases}$$

$$\mathcal{E}_B(b, pts, *\mathtt{x}) = \begin{cases} \bot, & \text{if } b(\mathtt{x}) \text{ is undefined, not a location, or out of bounds} \\ \widehat{Vals}, & \text{if } b(l) \text{ is undefined for some } l \text{ in } pts(\mathtt{x}), \text{ where } \alpha(l) = b(\mathtt{x}) \\ \{b(l) \mid pts(\mathtt{x}, l),\ \alpha(l) = b(\mathtt{x})\}, & \text{otherwise} \end{cases}$$

$$\mathbf{post}_B(b, pts, p, \mathtt{x} \ \mathtt{:=}\ E) = \begin{cases} \mathcal{L} \times B, & \text{if } \mathcal{E}_B(b, pts, E) = \bot \\ \{(p, b[\mathtt{x} \mapsto \hat{v}]) \mid \hat{v} \in \mathcal{E}_B(b, pts, E)\}, & \text{otherwise} \end{cases}$$

$$\mathbf{post}_B(b, pts, p, *\mathtt{x} \ \mathtt{:=}\ E) = \begin{cases} \mathcal{L} \times B, & \text{if } b(\mathtt{x}) \text{ is undefined, not a location, or out of bounds; or if } \mathcal{E}_B(b, pts, E) = \bot \\ \{(p, b[l \mapsto \hat{v}]) \mid pts(\mathtt{x}, l),\ \alpha(l) = b(\mathtt{x}),\ \hat{v} \in \mathcal{E}_B(b, pts, E)\}, & \text{otherwise} \end{cases}$$

Fig. 6. Abstract interpretation over B

Corollary 1. *If a points-to analysis $\mathcal{Q}$ is* SafeDeref*-sound, the composed memory safety analysis $\widetilde{\mathcal{B}} \circ \mathcal{Q}$ is* SafeDeref*-sound.*

Combining Corollary 1 with Theorems 3 and 4, we can compose $\widetilde{\mathcal{B}}$ with any of the points-to analyses described in Section 4 and the resulting analysis will be SafeDeref-sound. Recall from Theorem 1 that SafeDeref-soundness guarantees the detection of error states. If any non-SafeDeref state exists in $[\![\mathcal{P}]\!]_{\mathcal{C}}$, then a non-SafeDeref state is represented by the composed semantics; if only SafeDeref states are reachable in the composed analysis then no concrete non-SafeDeref state is reachable—the absence of error states can be proved.

6 Related Work

Methods for combining analyses have been described in the abstract interpretation community, starting with Cousot and Cousot [7]. The focus has been on exploiting mutual refinement to achieve the most precise combined analyses, as in Gulwani and Tiwari [19] and Cousot et al. [9]. The power domain of Cousot and Cousot [7, §10.2] provides a general model for analyses with conditional semantics. We believe our notion of conditional soundness provides a simpler model which captures the behavior of a variety of interesting analyses.

Pointer analysis for C programs has been an active area of research for decades [21,16,28,3,27,17,12,20,23]. The correctness arguments for points-to algorithms are typically stated informally—each of the analyses has been developed for the purpose of program transformation and understanding, not for use in a sound verification tool. Although Hind [21] proposes the use of pointer analysis in verification, the authors are not aware of any prior work that formally addresses the soundness of verification using points-to information.

Adams et al. [1] explored the use of Das' algorithm to prune the search space for a typestate checker and to generate initial predicates for a software model checker. In both cases, the use of the points-to information is essentially heuristic—the correctness of the overall approach does not depend on the points-to analysis being sound.

Dor, Rodeh, and Sagiv [15] describe a variation on traditional points-to analyses intended to improve precision for a sound, inter-procedural memory safety verifier. A proof of soundness is given in Dor's thesis [14]. However, the proof is not explicit about the obligations of the points-to analysis. We provide a more general framework for reasoning about verification using conditionally sound information.

Bruns and Chandra [4] provide a formal model for reasoning about pointer analysis based on transition systems. The focus of their work is primarily complexity and precision, rather than soundness.

Dhurjati, Kowshik, and Adve [13] define a program transformation which preserves the soundness of a flow-insensitive, equality-based points-to analysis (e.g., those of Steensgaard [27] and Lattner [23]) even for programs with memory safety errors. The use of an equality-based analysis is necessary to achieve an efficient implementation, but it limits the use of the technique in applications where a more precise analysis may be necessary, e.g., in verification. The soundness results we describe here are equally applicable to flow-sensitive, flow-insensitive, equality-based and subset-based pointer analyses.

Our abstraction for memory safety analysis is very similar to the formal models used in CCured [26] and CSSV [15]. Miné [25] describes a combined analysis for embedded control systems which incorporates points-to information. His analysis makes implementation-specific (i.e., unsound in general) assumptions about the layout of memory.

7 Conclusion

This work grew out of a simple, but puzzling question: is it possible to utilize the results of an analysis (points-to) whose soundness is dependent on a property (memory-safety) in a sound analysis for the same property? There seemed to be a circularity that could make a sound combination impossible.

Studying this question, we were led to a more precise statement of the soundness properties of points-to analysis and to the definition of conditional soundness. The final result shows that the combination is sound enough to correctly prove the absence of errors, although it may not be strong enough to point out every possible error.

We have concentrated here on points-to and memory safety analysis, but our conditional soundness framework is by no means restricted to these domains. For example, some static analyses are sound only assuming sequential consistency, that integer overflow does not occur, or that the program is free of floating point exceptions. The soundness claims of such analyses could be refined using the methods we have described in this paper.

Acknowledgments. This material is based upon work supported by the National Science Foundation under Grant No. 0341685. Additional support was provided by NSF Grant No. 0644299.

References

1. Adams, S., Ball, T., Das, M., Lerner, S., Rajamani, S.K., Seigle, M., Weimer, W.: Speeding up dataflow analysis using flow-insensitive pointer analysis. In: Static Analysis Symposium, Madrid, Spain, pp. 230–246 (September 2002)
2. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading (1988)
3. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (May 1994)
4. Bruns, G., Chandra, S.: Searching for points-to analysis. In: Foundations of Software Engineering, Charleston, South Carolina, pp. 61–70 (November 2002)
5. Conway, C.L., Dams, D., Namjoshi, K.S., Barrett, C.: Points-to analysis, conditional soundness, and proving the absence of errors. Technical Report TR2008-910, New York University, Dept. of Computer Science (2008)
6. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages, Los Angeles, California, pp. 238–252 (1977)
7. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Principles of Programming Languages, San Antonio, Texas, pp. 269–282 (1979)
8. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: European Symposium on Programming, Edinburgh, Scotland, pp. 21–30 (April 2005)
9. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of abstractions in the ASTRÉE static analyzer. In: Asian Computing Science Conference (ASIAN), Tokyo, Japan (December 2006)
10. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Principles of Programming Languages, Tucson, Arizona (January 1978)
11. Dams, D., Namjoshi, K.S.: Orion: Building blocks for program analyzers. In: Formal Methods for Components and Objects, Amsterdam, The Netherlands (November 2005)
12. Das, M.: Unification-based pointer analysis with directional assignments. In: Programming Language Design and Implementation, Vancouver, British Columbia, pp. 35–46 (2000)
13. Dhurjati, D., Kowshik, S., Adve, V.: SAFECode: enforcing alias analysis for weakly typed languages. In: Programming Language Design and Implementation, Ottawa, Canada, pp. 144–157 (June 2006)
14. Dor, N.: Automatic Verification of Program Cleanness. PhD thesis, Tel Aviv University (December 2003)
15. Dor, N., Rodeh, M., Sagiv, M.: CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In: Programming Language Design and Implementation, San Diego, California, pp. 155–167 (July 2003)
16. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Programming Language Design and Implementation, pp. 242–256 (June 1994)

17. Foster, J.S., Fähndrich, M., Aiken, A.: Flow-insensitive points-to analysis with term and set constraints. Technical Report UCB/CSD-97-964, University of California, Berkeley (August 1997)
18. Ghiya, R., Lavery, D.M., Sehr, D.C.: On the importance of points-to analysis and other memory disambiguation methods for C programs. In: Programming Language Design and Implementation, Snowbird, Utah, pp. 47–58 (June 2001)
19. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: Programming Language Design and Implementation, Ottawa, Canada (June 2006)
20. Heintze, N., Tardieu, O.: Demand-driven pointer analysis. In: Programming Language Design and Implementation, Snowbird, Utah, pp. 24–34 (June 2001)
21. Hind, M.: Pointer analysis: Haven't we solved this problem yet? In: Program Analysis for Software Tools and Engineering, Snowbird, Utah (June 2001)
22. ISO Standard - Programming Languages - C, ISO/IEC 9899:1999 (December 1999)
23. Lattner, C.: Macroscopic Data Structure Analysis and Optimization. PhD thesis, University of Illinois at Urbana-Champaign (May 2005)
24. Lerner, S., Grove, D., Chambers, C.: Composing dataflow analyses and transformations. In: Principles of Programming Languages, Portland, Oregon, pp. 270–282 (2002)
25. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: Languages, Compilers, and Tools for Embedded Systems, Ottawa, Canada (2006)
26. Necula, G.C., McPeak, S., Weimer, W.: CCured: type-safe retrofitting of legacy code. In: Principles of Programming Languages, Portland, Oregon, pp. 128–139 (January 2002)
27. Steensgaard, B.: Points-to analysis in almost linear time. In: Principles of Programming Languages, St. Petersburg Beach, Florida, pp. 32–41 (January 1996)
28. Wilson, R.P., Lam, M.S.: Efficient context-sensitive pointer analysis for C programs. In: Programming Language Design and Implementation, San Diego, California, pp. 1–12 (June 1995)

Protocol Inference Using Static Path Profiles

Murali Krishna Ramanathan[1], Koushik Sen[2], Ananth Grama[1], and Suresh Jagannathan[1]

[1] Department of Computer Science, Purdue University
{rmk,ayg,suresh}@cs.purdue.edu
[2] Electrical Engineering and Computer Science, University of California, Berkeley
ksen@eecs.berkeley.edu

Abstract. Specification inference tools typically mine commonalities among states at relevant program points. For example, to infer the invariants that must hold at all calls to a procedure p requires examining the state abstractions found at all call-sites to p. Unfortunately, existing approaches to building these abstractions require being able to explore all paths (either static or dynamic) to all of p's call-sites to derive specifications with any measure of confidence. Because programs that have complex control-flow structure may induce a large number of paths, naive path exploration is impractical.

In this paper, we propose a new specification inference technique that allows us to efficiently explore statically all paths to a program point. Our approach builds *static path profiles*, profile information constructed by a static analysis that accumulates predicates valid along different paths to a program point. To make our technique tractable, we employ a summarization scheme to merge predicates at join points based on the frequency with which they occur on different paths. For example, predicates present on a *majority* of static paths to *all* call-sites of any procedure p forms the pre-condition of p.

We have implemented a tool, MARGA, based on static path profiling. Qualitative analysis of the specifications inferred by MARGA indicates that it is more accurate than existing static mining techniques, can be used to derive useful specification even for APIs that occur infrequently (statically) in the program, and is robust against imprecision that may arise from examination of infeasible or infrequently occurring dynamic paths. A comparison of the specifications generated using MARGA with a dynamic specification inference engine based on CUTE, an automatic unit test generation tool, indicates that MARGA generates comparably precise specifications with smaller cost.

1 Introduction

The importance of clearly defined specifications to software development, maintenance, and testing is well-understood. Model-checking [5,14,26], type systems [7,9,10], typestate interpretation [17,13] and related static analyses [26]

© Springer-Verlag Berlin Heidelberg 2008

are valuable only when proper specifications are available. The absence of specifications can also lead to improper reuse of program components and weaken the effectiveness of testing mechanisms[11,12,22].

In some cases, specifications are relatively easy to express (e.g., procedure p must always be called after data structure d is initialized), or are well-documented (e.g., a call to `socket` must be present before a call to `bind` and the return value of `socket` must be checked for erroneous conditions). In many cases, however, specifications are unknown, and even when available, are often informal.

To remedy this issue, there has been much recent interest in devising techniques that automatically extract program properties by mining program behavior. The effectiveness of mining critically depends on a sufficient number of use cases that can be examined. For example, if we are interested in inferring the pre-conditions that must hold prior to a call to a procedure p, we benefit by examining multiple calls to p. The more calls analyzed, the greater the likelihood we can effectively distinguish between predicates present at these calls that are truly part of p's specification from those that, although present, are nonetheless irrelevant. In general, the confidence in a mined pre-condition is significantly higher if it is observed in 90,000 out of 100,000 occurrences, compared to when it is observed in 9 out of 10 occurrences, even though the underlying percentage of occurrences are the same [15].

With a sufficient number of test cases, dynamic mining techniques can generate execution traces which contain a potentially large number of calls to p. Unfortunately, the cost to generate these traces may be high if we wish to ensure that these traces define a comprehensive enumeration of all possible executions of the program [11,12,22]. On the other hand, static techniques can only rely on static properties to identify call points; for example, if a call to p occurs at m different static program points in the source, only the predicates present at those m points can be mined. On the benchmarks used in our study, we observed that on an average, 80-85% of the procedures in these benchmarks are not invoked more than five times statically.

Furthermore, existing static techniques do not take into account the number of paths leading to different call-sites of the same procedure. Consider a predicate π that occurs at one call-site c_1, but which is absent at another call-site c_2 of the same procedure p. Static inference techniques would naturally deduce that π is not part of p's pre-condition. However, the number of paths leading to c_1 may significantly be greater than c_2 (e.g., c_2 may be part of an infrequently occurring error-inducing path). Indeed, it may be precisely the absence of π at c_2 that leads to an error.

The underlying premise of our work is that we can effectively apply the benefits of a dynamic analysis (i.e., generating a desired quantity of data for the purposes of mining) to a static specification mining algorithm. However, exploring all paths and generating the traces associated with each path statically has two significant disadvantages: (a) there are an exponential number of paths that would need to be examined, and (b) if only a subset of all paths are explored, then this approach has the same disadvantage of incompleteness common to any dynamic mining strategy.

Our main contribution in this paper is the development of an intelligent comprehensive path enumeration and summarization scheme that does not lead to exponential time and space costs. This goal is achievable because we are interested in deriving properties that are not path specific, but merely valid over a majority of the paths examined.

We define an inter-procedural, path-based static analysis that collects a set of program predicates that define potential pre-conditions to procedure calls. If procedure p has pre-condition π, it means that that all the predicates comprising π should hold before any call to p. These predicates can encode control flow (e.g., a call to `bind` must be preceded by a call to `socket`) as well as dataflow properties (e.g., the return value of `socket` is always validated before a call to `bind`). To compute pre-conditions, we analyze the predicates present along each control flow edge in the program's control-flow graph. At any join point j, where multiple paths merge, we keep track of the *number* of paths, n_j, leading to the join point and the number of times a predicate π is valid on these n_j paths. This information, which we refer to as a *static path profile*, is transferred to outgoing edges, and the process repeated until all control flow edges are traced. We use procedure summaries to make the approach scalable for inter-procedural analysis. To compute the pre-conditions of a procedure p, we take the cumulative sets of predicates associated with its different call-sites and their associated path profiles to derive the required specifications.

We have implemented a tool, MARGA, for computing path profiles for C programs. Note that an essential assumption underlying our approach is that the probability of occurrence of a dynamic path is likely to be equal to that of a static path. Clearly, such an assumption need not hold in general. Static paths may be infeasible, i.e., not be traversable under any dynamic execution. Similarly, a path that occurs frequently statically may occur infrequently dynamically because there may be stringent runtime conditions that dictate when the path can be traversed that are not captured by a static analysis. Conversely, a path that occurs frequently dynamically (e.g. the back edge of a long-lived loop) may occur infrequently statically. Fortunately, for the purposes of specification inference, we demonstrate that static path profiling is robust against inaccuracies introduced by failing to recognize (statically) infeasible, or infrequently occurring static and dynamic paths.

To support this claim, we have also implemented a dynamic specification inference engine that mines comprehensive dynamic executions of the program generated by CUTE [22], an automatic test generation tool. A comparison of the specifications inferred by MARGA and the dynamic inference engine reveals that infeasibility of program paths (or lack of correlation between probabilities of static *vs.* dynamic paths) has little impact on the quality of the specifications generated.

Bugs in programs present another challenge to specification inference since they may invalidate correct predicates from a specification or introduce incorrect ones. Test generation tools can help identify commonly occurring bugs since such bugs by definition must occur on many dynamic paths. Because these bugs can be fixed, we assume programs are *mostly* free of errors. Bugs that are found on infrequently occurring dynamic paths are not always captured by unit testing. However, the paths on which these bugs occur must therefore necessarily correspond to profiled static paths with small weights. Consequently, the influence of these bugs on derived specifications is small.

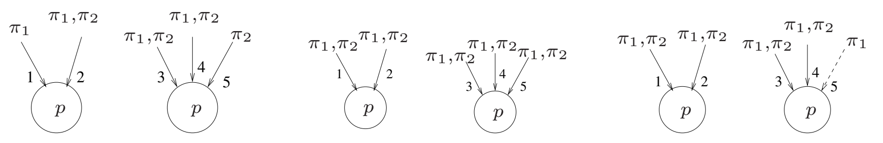
Our experimental results using MARGA show that the analysis (a) can effectively infer specifications even for procedures with a small number of statically apparent call-sites; (b) exhibits fewer false negatives compared to static specification inference techniques that do not take path profiles into consideration, and (c) displays precision closer to that of an exhaustive dynamic path exploration technique.

2 Motivation

Dynamic specification inference techniques suffer from the problem of *under approximation* i.e., a set of predicates $\Theta_u = \{\pi_1, \pi_2, \ldots \pi_n\}$ is declared to hold before a call to procedure p even when only a subset of the predicates found in Θ_u are valid elements of p's pre-condition set; this is possible because not all possible paths to the call may have been examined, and these unexamined paths may invalidate the inclusion of some of the π_i in Θ_u.

Similarly, static specification inference techniques suffer from the problem of *over approximation*, i.e., a set of predicates Θ_o is considered to hold before a call to procedure p, even though other predicates (not present in Θ_o) should also be included; this is possible because a particular predicate that cannot be proven to hold along a certain path may result in its omission from Θ_o, even if that path is infeasible (e.g., the path follows a branch that could never be taken) or erroneous.

We elaborate on these points using the example shown in Figure 1. Before every call to procedure p, there are certain predicates that hold. For example, in Figure 1(a), there are two call-sites to p. There are two paths, labeled `1` and `2`, to one of the call-sites; on path `1`, predicate π_1 holds and on path `2`, predicates π_1 and π_2 hold. There are three paths leading to the other call-site to p (the call-site on the right of Figure 1(a)); these paths are labeled `3, 4` and `5`, with predicates π_1 and π_2 valid on paths `3` and `4`, and π_2 valid on path `5`. In a dynamic analysis scheme, if the paths `2, 3` and `4` are the only ones traversed, we may erroneously conclude that both π_1 and π_2 hold always before a call to procedure p. Note that this case is difficult to distinguish from the scenario



(a) Under Approximation (b) Correct Pre-conditions (c) Over Approximation

Fig. 1. An example illustrating under- and over-approximation of predicates

```
399 add_listen_addr(ServerOptions *options, char *addr, u_short port)
      ...
403   if (options->num_ports == 0)
404     options->ports[options->num_ports++] = SSH_DEFAULT_PORT;
      ...
407   if (port == 0)
408    for (i = 0; i < options->num_ports; i++)
409     add_one_listen_addr(options, addr, options->ports[i]);
410   else
411     add_one_listen_addr(options, addr, port);
```

Fig. 2. Motivating Example for over-approximation from `openssh-4.2p1`

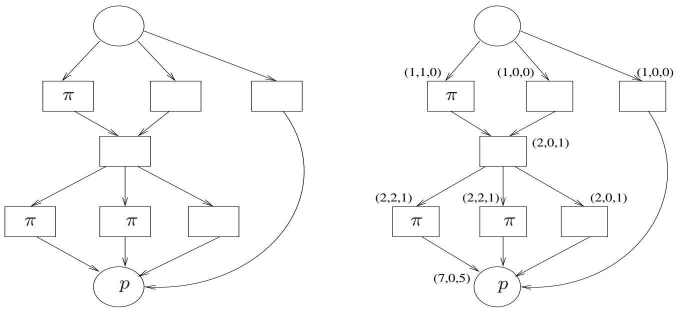
illustrated in Figure 1(b) where π_1 and π_2 *indeed* form the precondition for p. Ensuring that the paths `1` and `5` in Figure 1(a) are traversed depends upon the comprehensiveness of the test suite.

The problem of *over approximation* is illustrated in Figure 1(c). Here, there is one infeasible path (path `5`) to a call-site of p. A typical static analysis would conclude that one call to p (through paths `1` and `2`) has a set of predicates that include π_1 and π_2. Because of the absence of π_2 on the infeasible path `5`, the analysis would conclude that the other call to p (accessed through paths `3, 4` and `5`) does not include π_1 and π_2. Thus, given only two (static) calls to p, no statistically meaningful determination of p's pre-conditions can be made.

We provide further motivation for over approximation using a real-world example – statically deriving a specification for the `bind` system call in the Linux `socket` library. Part of the documented specification is that the address (second parameter to `bind`) corresponds to a specific address family (e.g., AF_UNIX, AF_INET). There are eight call-sites of `bind` in `openssh-4.2p1` of which *all* paths to five of the call-sites satisfy this specification. However, as shown in Figure 2, there exists a path to one of the call-sites where the address family is not set (`add_one_listen_addr` is not invoked). This happens when both `port` and `options->num_ports` are 0. This path is infeasible since both `port` and `options->num_ports` cannot be 0 simultaneously (line 404). Nevertheless, without the assistance of a theorem prover, static mining implementations must conservatively conclude that this is indeed a feasible path, and thus would be unable to conclude that the address family must be set prior to the call to `bind`. Using static path profiles, on the other hand, we will correctly weight the likelihood of this path occuring, and will not take into serious consideration the absence of the assignment to the address family.

3 Deriving Specifications

A simple technique to derive specifications is to trace each path in the program and then infer the set of valid pre-conditions from the traced paths. Consider the example shown in Figure 3(a). There are seven paths on total to a call-site



(a) Full Path Exploration (b) Static Path Profiles

Fig. 3. Illustrative example. Rectangles indicate predicates, circles indicate call-sites. Empty rectangles/circles indicate arbitrary predicates/procedure calls.

of some procedure p. If every path is traced statically, it is clear that among five out of the seven paths, the predicate π holds and is a precondition for p with confidence 71.42%. Although this scheme is simple, the cost associated with tracing each path is exponential in the number of edges in the program.

The key insight to our approach is that obtaining aggregate information associated with multiple paths is sufficient for generating interesting pre-conditions. Knowing the specific paths in which π holds is uninteresting from the perspective of specification inference. A static path profile is the cumulative information of predicates that hold across all possible paths to a specific call-site.

Path information is collected by examining the program's control-flow graph. Each node v in the CFG is annotated as a three tuple (n_v, m_v, q_v) for every predicate π under consideration, where the definition of the tuple components is as follows:

– n_v is the total number of paths leading to v,
– $m_v = \begin{cases} n_v, \text{if predicate } \pi \text{ holds at } v \\ 0 \text{ otherwise} \end{cases}$
– $q_v = \Sigma_u \, \texttt{max}\,(m_u, q_u)$ where $u \in$ predecessor(v) in the CFG.

At any given node, we can derive the number of paths any predicate π holds by observing the three-tuple (n_v, m_v, q_v) associated with the predicate π at that node. The number of predicates examined at a node is directly proportional to the number of variables. Intuitively, q_v specifies the number of paths through v in which the predicate is valid. If a predicate π is valid on some number of incoming paths *upto* node v, but in addition also happens to be asserted *at* v, it is clear π holds on all paths *through* v $(m_v = n_v)$. The nodes downstream from v decide the number of paths on which π holds using q_v and m_v. If the number of paths for which π holds is i, then $i = \texttt{max}\,(m_v, q_v)$; the fact that predicate π holds on i paths is denoted as π_i.

For example, consider the annotated graph counterpart of Figure 3(a) in Figure 3(b) associated with predicate π. Let one of the two nodes annotated

`(2,0,1)` be v. This annotation denotes the fact that there are two possible paths to node v, predicate π is not explicitly valid at v, and the total number of paths on which π is valid is one (written π_1).

Loops pose complications for building path profiles because they represent a potentially infinite set of executions. To make our approach tractable, we perform a simple fixpoint calculation to compute the path profile for back-edges in loops. Initially, we assume the back-edge does not contribute to the profile weights of any path found within the loop. In subsequent iterations of the analysis, the back edge on the loop contributes exactly once to the profile weights, albeit with the predicates being derived propagated through the back edge multiple times. Since the computation of the tuple is monotonic (since q_v computes the maximum of the profiles of its predecessors which is bounded by the number of paths in the loop body that include the back-edge), the analysis is guaranteed to converge.

The annotation marking mechanism must also take into account nodes in the control-flow graph that represent call-sites (e.g., the node labeled p). Path profiles distinguish between incoming and outgoing annotations. The incoming annotation in p's case is `(7,0,5)`. Incoming annotations are used to generate preconditions for p. Thus, to infer the pre-condition for p requires no inspection of p's body. In this case, π_5 holds true at node p, i.e., five paths of the total set of paths have π to be true. Outgoing annotations (not shown in the figure), on the other hand, capture path profile summary information. The summary information for some procedure p gives the number of paths within p for which the predicate holds upon exit from p, which in turn is given by the annotation at p's *return* node. Summary information is used to define incoming annotations for other call-sites downstream in the graph. We elaborate on this point in the next section.

4 MARGA : Implementation Details

We have implemented a tool name MARGA based on the above approach. It takes as input the program source and a user-defined confidence threshold for determining when a predicate should form part of a pre-condition, and produces as output pre-conditions (i.e., a set of predicates) for every procedure. These pre-conditions indicate the conditions that must hold prior to any call of the associated procedure.

We first generate the control-flow graph for each procedure using Codesurfer [3]. The resulting graph is processed using the algorithm given in Figure 4. The number of paths to each node in the graph is first computed. Subsequently, the q value for the node is computed for each predicate by considering all its parent nodes. If the node is a call-site, then the *procedure summary* associated with that call is also added to the set of predicates that will flow into other adjacent nodes in the graph. The procedure summary is the summary of predicate information along with the total number of paths and the number of paths for which each predicate holds at the *return* node of the procedure. This process is repeated until a convergent path profile for the loop back-edge is

```
procedure BUILDPREDICATES
  ▷ Input: G(V,E) , directed, acyclic CFG of $\alpha$; V is topologically sorted;
  ▷ Output: Annotated Graph G
  ▷ Auxiliary Information:
      predicates($u$): predicates generated at $u$; flow($u$): set of predicates valid at $u$;
      precond($u$): set of predicates that are used for generating preconditions associated with procedure at $u$;
      CALLSITE($u$): true if $u$ is a callsite; RETURN($u$): true if $u$ is the return node from procedure $\alpha$;

1  $iterate \leftarrow$ true
2  while $iterate$ do
3    iterate $\leftarrow$ false
4    for each node $u = 1$ to $|V|$
5      $oldflow \leftarrow$ flow($u$)
6      for all predecessors $v$ of $u$
7        $n_u \leftarrow n_u + n_v$
8        flow($u$) $\leftarrow$ flow($u$) $\cup$ predicates($v$)
9        for each predicate $\pi$ in flow($v$)
10         $q_u(\pi) \leftarrow q_u(\pi) + \max(m_v(\pi), q_v(\pi))$
11         $m_u(\pi) \leftarrow 0$
12     flow($u$) $\leftarrow$ flow($u$) $\cup$ predicates($u$)
13     for each predicate $\pi$ in data_predicate($u$)
14       $m_u(\pi) \leftarrow n_u$
15     if CALLSITE($u$) is true then
16       precond($u$) $\leftarrow$ flow($u$)
17       flow($u$) $\leftarrow$ flow($u$) $\cup$ proc_summary[func($u$)]
18     if RETURN($u$) is true then
19       proc_summary[$\alpha$] $\leftarrow$ flow($u$)
20     if $oldflow \neq$ flow($u$) then $iterate \leftarrow$ true
```

Fig. 4. Algorithm for building predicates

```
procedure GETPRECONDITIONS
    ▷ Input: $\alpha$: a procedure in the program;
             C $= \{c_1, c_2, ...c_n\}$ is the set of call sites of $\alpha$;
             $\beta$ = user-defined threshold for generating preconditions
    ▷ Output:set of preconditions for $\alpha$
1   for each node $c_i$
2       for each predicate $\pi$ in precond($c_i$)
3           $q_t(\pi) \leftarrow q_t(\pi) + q_{c_i}(\pi)$
3           $n_t(\pi) \leftarrow n_t(\pi) + n_{c_i}$
4       flow_t $\leftarrow$ flow_t $\cup$ precond($c_i$)
5   for each predicate $\pi$ in flow_t
6       if $\frac{q_t}{n_t} > \beta$
7           preconditions[$\alpha$] $\leftarrow$ preconditions[$\alpha$] $\cup \{\pi\}$
```

Fig. 5. Generate preconditions

computed.Yet another fix point iteration is performed to ensure that dependencies crossing procedure boundaries (as given by the *procedure summary* and *return*) are completely captured.

At the end of the fixed point calculation, the algorithm shown in Figure 5 is executed to obtain the pre-conditions associated with each procedure in the program. To generate the pre-condition for a procedure p, each call-site of p is considered. The predicates that can be used in computing the preconditions from each of these call-sites is extracted and the total number of paths in which the predicate holds (q_t) across all call-sites is computed. Similarly, the total number of paths to all call-sites (n_t) is also calculated. If the ratio of the number of

Table 1. Benchmark Information

Source	Version	LOC	CFG nodes	Procedure count	Avg. paths to call-sites	Total Specifications	Analysis time (s)
`zebra`	0.95a	183K	145K	3342	4721.64	1323	555
`apache`	2.2.3	273K	102K	2079	7281.30	676	561
`openssh`	4.2p1	68K	88K	1281	7175.75	619	501
`osip2`	3.0.1	24K	34K	666	4028.32	158	104
`procmail`	3.22	9K	16K	298	33696.15	120	297
`buddy`	2.4	10K	10K	173	5653.23	133	140

paths on which a predicate holds compared to the total number of paths at all call-sites is greater than β, a user-defined threshold, the predicate is added to the pre-condition for p.

5 Experiments

We validate the idea of using static path profiles on selected benchmark sources to demonstrate scalability and effectiveness. We extract pre-conditions for six sources: `apache`, `buddy`, `zebra`, `openssh`, `osip2`, and `procmail`. Specific details about these benchmarks are provided in Table 1. The size of the benchmarks varies from 9K to 273KLOC. Since default configurations are used to compile these sources, we believe that the number of control flow nodes represents a more reliable indicator of effective source size than lines of code. The number of control flow nodes ranges from 10K to 143K. We also present the number of user-defined procedures examined in the table.

We have implemented our tool in C++ and have performed our experiments on a Linux 2.6.11.10 (Gentoo release 3.3.4-r1) system running on an Intel(R) Pentium(R) 4 CPU machine operating at 3.00GHz, with 1GB memory. The time taken for performing the analysis is presented in Table 1.

5.1 Quantitative Assessment

We derive pre-conditions containing three different types of predicates – assignment, comparison and precedence. As their names suggest, assignment predicates reflect the assignment of values (or results of procedure calls) to variables; comparison predicates include six kinds of logical comparison operations ($>$, $<$, $\neq$, $=$, $\geq$ and $\leq$) between variables and/or constants; a precedence predicate is an ordered sequence of procedures whose calls must precede the call to the procedure being examined. The total number of pre-conditions generated for procedures, where the predicates are valid on at least 70% of the paths is given in Table 1. The size distribution (number of predicates within a pre-condition) for the generated pre-conditions is given in Figure 6. Among the generated pre-conditions, the size of the predicate set is less than two for a majority of the procedures. For example, observe that approximately 97% of the procedures in `apache` have

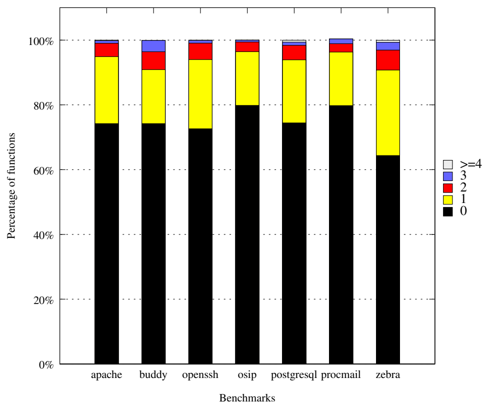


Fig. 6. Predicate distributions

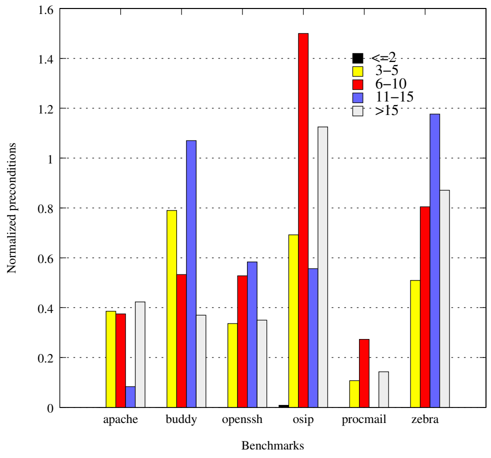


Fig. 7. Comparison with non-profile based inference

fewer than two predicates in their pre-conditions. The predicate size distribution display a similar pattern for different types of predicates.

We experimentally compare our approach with a non-profile based inference mechanism that does not leverage path profiles [21]. Briefly, the comparison metric is an analysis that requires a predicate to be satisfied along all paths to a call-site in order to be a valid candidate for inclusion as part of a procedure call's pre-condition. After accumulating the predicates at each call-site, we declare a predicate as a pre-condition, if the predicate is valid in at least the user-defined threshold percentage of call-sites. We use the same user-defined threshold (70%) in deriving the predicates, i.e., if a predicate is valid in seven out of 10 call-sites, we declare the predicate as a pre-condition in the non-profile based inference scheme. In the path-profiling approach, we declare a predicate to be a pre-condition if it is valid in 70% of the *paths* to call-sites of the procedure.

Figure 7 presents the percentage of pre-conditions derived by non-profile based specification inference as compared to those derived using static path profiling. For example, for procedures with three to five call-sites in `zebra`, the former discovers roughly only half the predicates discovered by the path profile analysis. As expected, for procedures with fewer than three call-sites, the non-profile based inference scheme is not able to derive any pre-conditions. For example, in the case of `openssh`, no pre-condition is derived for procedures that have fewer than three statically apparent call-sites.

We also observe that in many cases, the set of pre-conditions generated with the non-profile based inference is a proper *subset* of the pre-conditions generated using the static path profiling approach. This is consistent with our expectation that typical static analyzes can lead to over approximation by eliminating valid predicates from pre-conditions. In some cases, however, such as `osip` or `zebra`, this hypothesis does not hold. Path profiling weights predicates based on the number of paths on which they hold across all call-sites. Consider a predicate π which occurs on k paths at n call-sites to procedure p. Suppose that paths are not evenly distributed among these n call-sites. If on m call-sites, π occurs on all paths, and m is greater than the threshold cutoff, the non-profile based inference will record π as a valid pre-condition. However, if the number of paths that flow into these m call-sites is much less than k, then the path profile analysis will nonetheless not include π as part of p's pre-conditions. In other words, a predicate that does not hold on a majority of paths may still hold on the paths to a majority of call-sites. Path profiling thus provides finer control over both the inclusion and exclusion of predicates than non-profile based inference.

5.2 Qualitative Assessment

We want to identify the impact of infeasible paths and approximations introduced by static path weights in the program on the quality of the specifications inferred. To do so, we compare our approach with a dynamic inference mechanism. Rather than using an existing test-suite to generate dynamic traces, we use CUTE, an automatic test generation tool, that provides extended coverage of the program, and thus helps reduce the possibility of under-approximation (compared to other dynamic analysis systems) as described in Section 2.

The test generation process initially runs with some random input and collects constraints $\{C_1, ..., C_{k-1}, C_k\}$ symbolically along the execution. To explore a previously unexplored path, a new input is generated that satisfies the constraints $\{C_1, ...C_{k-1}, \neg C_k\}$. If this path was explored earlier, then the set of constraints $\{C_1, ...\neg C_{k-1}\}$ is used to explore a different path. This process repeats until all paths in the program are explored. There are several issues that must be handled by the input generation process. Most importantly, when it becomes difficult to reason with symbolic constraints, concrete values from the program execution replace symbols to ensure progress of the test input generation process. We refer the reader to [22] for a more detailed description.

We track the predicates along different execution paths of the program and for each call to a procedure, group the set of predicates that precede it from the

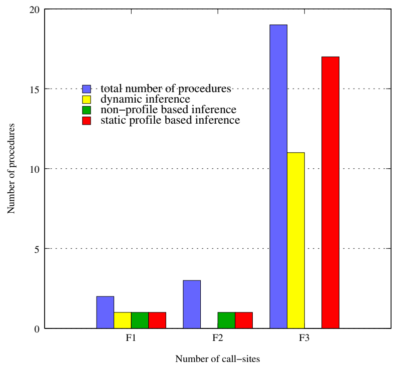


Fig. 8. Correctness of different inference schemes

start of the execution. Thus, across multiple executions, we would generate many such groups of predicates. To generate the precondition for the procedure, we apply frequent item-set mining [6]. The frequently occurring predicates across all the groups form the precondition.

We performed our comparison on `buddy`, an open source package that implements operations over Binary Decision Diagrams (BDD). We ran CUTE for a bounded number of iterations (1000), which took approximately 30 minutes, and in that process collected specifications for 24 procedures. Of these 24 procedures, only two procedures(F1) had more than 10 call-sites, three procedures (F2) had call-sites between five and 10 and the remaining 19 procedures(F3) had less than five call-sites. Using existing documentation, and manual inspection, we computed a reference specification for each of these procedures.

Figure 8 presents the results associated with our qualitative analysis. We applied three different schemes, (a) dynamic inference (b) non-profile based inference, and (c) path-profile based inference on this benchmark. For the set of two procedures in F1, all techniques provide similar precision and were able to detect preconditions correctly for one procedure. Under-approximation confounds the precision of dynamic inference for the set of three procedures in F2. The analysis for the procedures in F3 is more interesting. Because of the lack of frequency of call-sites for the procedures in this set, non-profile based static inference is ineffective in producing specifications with any degree of confidence. In contrast, path-profile based inference correctly identified the correct specification for 17 of the 19 procedures. Surprisingly, under-approximation still poses a problem even for a comprehensive test generation tool like CUTE; it failed to correctly generate specifications for 7 of the procedures that were successfully analyzed using static path profiling.

6 Related Work

Many interesting static mining approaches exist for specification inference. Kremenek *et al.* [16] develop a inference mechanism based on using factor graphs.

Even though, many useful specifications were generated, the approach requires either machine learning or user specifications to generate initial annotation probabilities, employs naming conventions for improving accuracy and is domain-specific. Ramanathan *et al.* [21] present an annotation-free approach to infer data flow specifications using frequent item-set mining and control flow specifications (precedence relations [20]) using sequence mining. This approach is path-sensitive, but does not take static path profiles into account: if a predicate does not hold at a majority of call-sites to a procedure, it is not included as part of the procedure's pre-condition. Shoham *e al.* [23] propose an approach for client-side mining of temporal API specifications based on static analysis. Li and Zhou present PR-Miner [18], a tool that relies on mining to identify frequently occurring program patterns. As this approach is not path-sensitive, spurious specifications can be generated even if a predicate holds in *at least* one path leading to a majority of call sites. Mandelin *et al.* [19] present a technique for synthesizing jungloid code fragments automatically based on the input and output types that describe the code and is useful for reusing existing code. An automatic specification mining technique that uses information about exceptions to identify temporal safety rules is presented in [25]. Because none of the above techniques performs mining on generated paths, the confidence in the derived specifications is statistically low. Due to the ability of our approach to simulate dynamic behavior and succinctly maintain data that covers all possible program paths, we are able to derive useful specifications with high confidence.

Ball and Larus [4] propose an approach for efficient path profiling. In their approach, dynamic runs of a program are profiled to gather information about different path executions. Recently, Vaswani *et al.* [24], present a scheme to identify a subset of interesting paths and use a compact numbering scheme using arithmetic coding techniques. Our approach is motivated by the algorithm presented by Ball and Larus [4] and is applied statically.

Ammons *et al.* [1] perform specification mining by summarizing frequent interaction patterns as state machines that capture temporal and data dependencies when interacting with API's or abstract data types. An approach to debug derived specifications using concept analysis is subsequently proposed by Ammons *et al.* in [2]. Daikon [8] is a tool for dynamically detecting invariants in a program. Yang *et al.* [27] present a technique for automatically inferring temporal properties by exploring event traces across versions of a program. All these approaches critically rely on test input providing comprehensive coverage. Our approach is independent of test inputs and covers all possible program paths.

Acknowledgements

This work is supported in part by the National Science Foundation under grant CNS-509387. We also like to thank the anonymous reviewers for their constructive feedback.

References

1. Ammons, G., Bodik, R., Larus, J.: Mining specifications. In: Proceedings of POPL 2002, pp. 4–16 (2002)
2. Ammons, G., Mandelin, D., Bodik, R., Larus, J.: Debugging temporal specifications with concept analysis. In: Proceedings of PLDI 2003, pp. 182–195 (2003)
3. Anderson, P., Reps, T., Teitelbaum, T.: Design and implementation of a fine-grained software inspection tool. IEEE Trans. on Software Engineering 29(8), 721–733 (2003)
4. Ball, T., Larus, J.: Efficient path profiling. In: MICRO-29 (December 1996)
5. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 103–122. Springer, Heidelberg (2001)
6. Burdick, D., Calimlim, M., Flannick, J., Gehrke, J., Yiu, T.: Mafia: A performance study of mining maximal frequent itemsets. In: FIMI 2003 (2003)
7. Chin, B., Markstrum, S., Millstein, T.: Semantic type qualifiers. In: Proceedings of PLDI 2005, pp. 85–95 (2005)
8. Ernst, M., Cockrell, J., Griswold, W., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE TSE 27(2), 1–25 (2001)
9. Foster, J., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: Proceedings of PLDI 2002 (2002)
10. Furr, M., Foster, J.: Checking type safety of foreign function calls. In: Proceedings of PLDI 2005 (2005)
11. Godefroid, P.: Compositional dynamic test generation. In: POPL 2007, pp. 47–54 (2007)
12. Godefroid, P., Klarslund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of PLDI 2005, Chicago, Il, pp. 213–223 (2005)
13. Henzinger, T., Jhala, R., Majumdar, R.: Permissive interfaces. SIGSOFT Softw. Eng. Notes 30(5), 31–40 (2005)
14. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2004)
15. Kapadia, A.S., Chan, W., Moye, L.A.: Mathematical Statistics With Applications. CRC, Boca Raton (2005)
16. Kremenek, T., Twohey, P., Back, G., Ng, A., Engler, D.: From uncertainty to belief: Inferring the specification within. In: Proceedings of OSDI 2006 (2006)
17. Lam, P., Kuncak, V., Rinard, M.: Generalized typestate checking for data structure consistency. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, Springer, Heidelberg (2005)
18. Li, Z., Zhou, Y.: Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In: Proceedings of ESEC-FSE 2005 (September 2005)
19. Mandelin, D., Xu, L., Bodik, R., Kimelman, D.: Jungloid mining: Helping to navigate the api jungle. In: Proceedings of PLDI 2005, pp. 48–61 (2005)
20. Ramanathan, M.K., Grama, A., Jagannathan, S.: Path-sensitive inference of function precedence protocols. In: Proceedings of ICSE 2007 (May 2007)
21. Ramanathan, M.K., Grama, A., Jagannathan, S.: Static specification inference using predicate mining. In: Proceedings of PLDI 2007, pp. 123–134 (2007)
22. Sen, K., Marinov, D., Agha, G.: Cute: A concolic unit testing engine for c. In: Proceedings of ESEC-FSE, pp. 263–272 (2005)

23. Shoham, S., Yahav, E., Fink, S., Pistoia, M.: Static specification mining using automata-based abstractions. In: ISSTA 2007: International Symposium on Software Testing and Analysis, pp. 174–184 (July 2007)
24. Vaswani, K., Nori, A.V., Chilimbi, T.M.: Preferential path profiling: compactly numbering interesting paths. In: Proceedings of POPL 2007, Nice, France (January 2007)
25. Weimer, W., Necula, G.: Mining temporal specifications for error detection. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 461–476. Springer, Heidelberg (2005)
26. Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In: Proceedings of POPL 2005 (2005)
27. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: Mining temporal api rules from imperfect traces. In: Proceedings of ICSE 2006 (May 2006)

Solving Multiple Dataflow Queries Using WPDSs

Akash Lal[1] and Thomas Reps[1,2]

[1] University of Wisconsin, Madison, Wisconsin, USA
{akash,reps}@cs.wisc.edu
[2] GrammaTech, Inc., Ithaca, NY, USA

Abstract. A dataflow query asks for the set of reachable (abstract) states, given a starting set of states. In this paper, we show how to optimize multiple queries on the same program (each with a different starting set of states) for better overall running time. After a preprocessing phase, we obtain an asymptotic improvement in answering dataflow queries. We use *weighted pushdown systems* as the abstract model of a program. Our techniques are interprocedural. They are general, yet provide an impressive speedup. We applied our algorithm to three very different applications, one based on finding affine relations using linear algebra, and others for model checking Boolean programs, and obtained 1.5-fold to 7-fold speedups.

1 Introduction

Dataflow analysis is concerned with approximating program behavior. A *dataflow query* asks for the set of (abstract) program states (forward or backward) reachable from a given starting set of states, where a state is a $(\text{program location}, \text{data store})$ pair. One common application of (forward) dataflow analysis is to pose a single dataflow query from the initial state in which program execution starts. This produces an over-approximation of all program states that may arise during its execution. However, in certain situations, multiple dataflow queries need to be posed on the same program, each with a different starting set of states.

One such need arises in the analysis of concurrent programs, in the method presented in [13], which tracks program evolution for a bounded number of context switches. Here, a concurrent program consists of a set of threads that communicate via shared memory. For a thread t of interest, the environment (consisting of the other threads) is only given control a fixed number of times. Each time, the environment can change the state of shared memory, thus affecting the execution of thread t. The analysis of such programs requires multiple dataflow queries to be posed on t. Whenever the environment changes the state of shared memory, a new query is posed on t, starting from this state

Multiple queries are also useful for program understanding, e.g., to find out the net effect of executing from one statement to another (to find dependences

© Springer-Verlag Berlin Heidelberg 2008

between them). Finding a loop summary for each loop is another example. Our applications (§5) are based on these examples.

Answering multiple queries on the same program independently from each other usually involves repeated work. In this paper, we do preprocessing to compute certain basic facts about the program that can be reused each time a new dataflow query is posed. This improves the running time needed for answering multiple dataflow queries on the same program.

At the intraprocedural level, this work is inspired by our previous result [9], where we showed how to use Tarjan's path sequence algorithm [23], which computes regular expressions to represent a set of paths in a graph, to obtain a faster algorithm for answering a single dataflow query. In this paper, we show that the information computed by Tarjan's algorithm is also useful to avoid having to *repeat* fixpoint computations for answering multiple queries.

At the interprocedural level, a set of program paths can no longer be captured with a regular expression (the set may be a context-free language). We develop new techniques to address this complication: we show what preprocessing can be done to avoid recomputation across procedure boundaries, and how to isolate the intraprocedural computation to be able to use our intraprocedural algorithm.

Overall, with our techniques, the preprocessing is quite efficient, usually faster than solving two dataflow queries. After preprocessing, we obtain asymptotic improvements in answering each dataflow query (for programs whose control structure is mostly reducible), and only require iteration to a fixpoint when the starting set of states is infinite (i.e., in other cases, we do not need to go around program loops or recursive procedures). Our experiments show that this approach is advantageous even if as few as two queries need to be answered.

Our approach applies to any dataflow-analysis problem in which one has a domain of distributive dataflow-transfer functions closed under composition [7,22]. Some examples can be found in [12,17,18]. This paper mainly presents the work using the framework of *weighted pushdown systems* (WPDSs) [18], which generalize previous work on interprocedural analysis frameworks [8,16,22]. For details on how variants of the technique can be incorporated in solvers that work over control-flow graphs (ICFGs) see [10].

The contributions of this paper can be summarized as follows:

– We show how information computed by Tarjan's path sequence algorithm can be used to obtain asymptotic improvements in answering multiple intraprocedural queries (§3).
– We give a new WPDS reachability algorithm for answering interprocedural queries that carries over the above asymptotic improvements (§4).
– We sketch variants of the technique that allow the ideas to be applied in other standard dataflow-analysis frameworks (§4).
– We applied our techniques to three applications (§5), and measured 1.5-fold to 7-fold speedups over previous techniques, including optimized ones [9].

The rest of the paper is organized as follows: §2 gives background on WPDSs. §3 presents our algorithm for the intraprocedural case, and §4 generalizes it to the interprocedural case (WPDSs). §5 reports experimental results. §6 discusses related work. Proofs and other details can be found in [10].

2 Program Model

Definition 1. *A* **pushdown system** *is a triple $\mathcal{P} = (P, \Gamma, \Delta)$ where P is the set of states or control locations, Γ is the set of stack symbols and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is the set of pushdown rules. A* **configuration** *of $\mathcal{P}$ is a pair $\langle p, u \rangle$ where $p \in P$ and $u \in \Gamma^*$. A rule $r \in \Delta$ is written as $\langle p, \gamma \rangle \hookrightarrow \langle p', u \rangle$ where $p, p' \in P$, $\gamma \in \Gamma$ and $u \in \Gamma^*$. These rules define a transition relation $\Rightarrow$ on configurations of $\mathcal{P}$ as follows: If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', u \rangle$ then $\langle p, \gamma u' \rangle \Rightarrow \langle p', uu' \rangle$ for all $u' \in \Gamma^*$. The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$.*

Without loss of generality, we restrict PDS rules to have at most two stack symbols on the right-hand side [21]. The standard approach for modeling program control flow with a PDS is as follows: $P = \{p\}$, Γ corresponds to program locations, and Δ corresponds to transitions in the interprocedural control-flow graph (ICFG)[1]: A CFG edge $u \rightarrow v$ is modeled by a PDS rule $\langle p, u \rangle \hookrightarrow \langle p, v \rangle$; A call to procedure g at location l that returns to r as $\langle p, l \rangle \hookrightarrow \langle p, g_{\text{enter}}\ r \rangle$; and a return from procedure g as $\langle p, g_{\text{exit}} \rangle \hookrightarrow \langle p, \varepsilon \rangle$. In such an encoding, a PDS configuration $\langle p, \gamma_1\ \gamma_2 \cdots \gamma_n \rangle$ stores the value of the program counter γ_1 and the stack of return addresses for unfinished calls as $\gamma_2, \gamma_3, \cdots, \gamma_n$ (in order).

A *weighted* PDS is obtained by supplementing a PDS with a weight domain that is a *bounded idempotent semiring* [2,18]. Such semirings are capable of encoding a number of abstractions [17]. WPDSs can encode the IFDS framework [16], and other dataflow analyses; see [9,18] for more details.

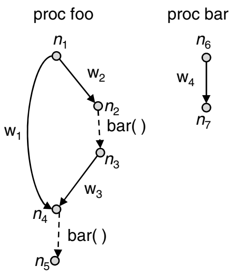
Definition 2. *A* ***bounded idempotent semiring*** *(or "weight domain") is a tuple $(D, \oplus, \otimes, \overline{0}, \overline{1})$, where D is a set of* **weights***, $\overline{0}, \overline{1} \in D$, and $\oplus$ (combine) and $\otimes$ (extend) are binary operators on D such that*

1. *$(D, \oplus)$ is a commutative monoid with $\overline{0}$ as its neutral element, and where $\oplus$ is idempotent. $(D, \otimes)$ is a monoid with the neutral element $\overline{1}$.*
2. *$\otimes$ distributes over $\oplus$, i.e., for all $a, b, c \in D$ we have*
$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \text{ and } (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)\ .$$
3. *$\overline{0}$ is an annihilator with respect to $\otimes$, i.e., for all $a \in D$, $a \otimes \overline{0} = \overline{0} = \overline{0} \otimes a$.*
4. *In the partial order $\sqsubseteq$ defined by $\forall a, b \in D$, $a \sqsubseteq b$ iff $a \oplus b = a$, there are no infinite descending chains.*

One may think of weights as dataflow transformers, extend as transformer composition, combine as *meet*, $\overline{0}$ as the transformer for an infeasible path, and $\overline{1}$ as the identity transformer.

The *height* $\mathcal{H}$ of a weight domain is defined to be the length of the longest descending chain in the semiring (if it exists). In this paper, we assume the

[1] An ICFG is a set of CFGs, one for each procedure, with additional edges going from a call-site to the entry node of the callee and from its exit node to the return site.



(1)	$\langle p, n_1\rangle \hookrightarrow \langle p, n_2\rangle$	w_2
(2)	$\langle p, n_1\rangle \hookrightarrow \langle p, n_4\rangle$	w_1
(3)	$\langle p, n_2\rangle \hookrightarrow \langle p, n_6\ n_3\rangle$	$\overline{1}$
(4)	$\langle p, n_3\rangle \hookrightarrow \langle p, n_4\rangle$	w_3
(5)	$\langle p, n_4\rangle \hookrightarrow \langle p, n_6\ n_5\rangle$	$\overline{1}$
(6)	$\langle p, n_6\rangle \hookrightarrow \langle p, n_7\rangle$	w_4
(7)	$\langle p, n_7\rangle \hookrightarrow \langle p, \varepsilon\rangle$	$\overline{1}$

Fig. 1. A program with two procedures and its corresponding WPDS. Procedure calls are represented using dashed arrows.

height to be finite for ease of discussing complexity results. (For cases when the height is unbounded, the value $\mathcal{H}$ in the complexity results can be interpreted as the length of the longest descending chain that occurs while solving a particular problem instance, which is always bounded.)

Definition 3. *A* **weighted pushdown system** *is a triple $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ where $\mathcal{P} = (P, \Gamma, \Delta)$ is a pushdown system, $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ is a bounded idempotent semiring and $f : \Delta \to D$ is a map that assigns a weight to each pushdown rule.*

Let $\sigma = [r_1, \ldots, r_k] \in \Delta^*$ be a sequence of rules. We define $v(\sigma) \overset{\text{def}}{=} f(r_1) \otimes \ldots \otimes f(r_k)$. Moreover, if for two configurations c and c' of $\mathcal{P}$, σ is a rule sequence that transforms c to c', we say $c \Rightarrow^{\sigma} c'$.

Definition 4. *Let $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$ be a WPDS, where $\mathcal{P} = (P, \Gamma, \Delta)$, and let $S, T \subseteq P \times \Gamma^*$ be sets of configurations. The* ***interprocedural meet-over-all-paths (IMOP)*** *value IMOP(S, T) is defined as $\bigoplus\{v(\sigma) \mid s \Rightarrow^{\sigma} t, s \in S, t \in T\}$.*

The IMOP value is the net transformation that occurs when going from one set of configurations to another. We write IMOP(s, t) for IMOP$(\{s\}, \{t\})$.

Fig. 1 shows how a program can be encoded using a WPDS. Each ICFG edge e is encoded as a PDS rule whose weight is the dataflow transformer for e. More details on encoding programs as WPDSs can be found in [18,17].

Model Semantics. In WPDSs, *program states* are represented using weighted configurations, which are configuration-weight pairs. The pair (c, w) describes the control state of the program as the PDS stack c, and the data state of the program using the weight w. A set of program states is represented by a function $\beta : P \times \Gamma^* \to D$, standing for the set $\{(c, \beta(c)) \mid c \in P \times \Gamma^*\}$. The set of all forward-reachable states starting from β is the set $\mathit{poststar}(\beta) = \{(c', \oplus_c\{\beta(c) \otimes \text{IMOP}(c, c')\}) \mid c' \in P \times \Gamma^*\}$. In this case, we say that configuration c' can be reached with weight $\mathit{poststar}(\beta)(c')$ (which is $\overline{0}$ if c' is not reachable).

For example, the initial state of the program in Fig. 1 is $(\langle p, n_1\rangle, \overline{1})$ and its reachable states include $(\langle p, n_6\ n_3\rangle, w_2)$ and $(\langle p, n_6\ n_5\rangle, w_1 \oplus (w_2 \otimes w_4 \otimes w_3))$.

3 Solving Multiple Intraprocedural Queries

Our interprocedural algorithm (§4) will need to solve multiple intraprocedural queries. Thus, we address the latter case first. A directed graph is a special case of a PDS (no call or return rules). When a weight domain is paired with a directed graph, we obtain a model for intraprocedural analysis. To simplify the discussion of intraprocedural algorithms, we specialize some definitions to weighted graphs.

Definition 5. *A* ***weighted graph*** *G is a tuple (V, E, λ), where (V, E) is a directed graph, and $\lambda : E \to D$ is a function that labels each edge with a weight.*

For vertices $v_1, v_2 \in V$, a path σ is defined as a sequence of edges that connect v_1 to v_2, in the standard way. In such a case, we say $v_1 \Rightarrow^{\sigma} v_2$. The weight of a path $\sigma = [e_1, e_2, \cdots, e_n]$, written as $\lambda(\sigma)$, is defined to be $\lambda(e_1) \otimes \lambda(e_2) \otimes \cdots \otimes \lambda(e_n)$. For sets of vertices $S, T \subseteq V$, the meet-over-all-paths (MOP) value is defined as the combine of weights of all paths that lead from a vertex in S to a vertex in T: $\text{MOP}(S, T) = \bigoplus\{\lambda(\sigma) \mid s \Rightarrow^{\sigma} t, s \in S, t \in T\}$. When $S = \{s\}$ and $T = \{t\}$ are singleton sets, we write $\text{MOP}[s, t]$ as a shorthand for $\text{MOP}(S, T)$.

Program states are vertex-weight pairs (vertices replace PDS configurations). Computing reachable states reduces to solving the following query:

Definition 6. *Given a weighted graph G, and a set of vertices $S \subseteq V$ with a weight assignment $\mu : S \to D$, the* **IntraQ**-***query*** *is to compute the weights $\text{IntraQ}_G(S, \mu)(v) = \bigoplus_{s \in S}\{\mu(s) \otimes MOP[s, v]\}$ for each $v \in V$.*

$\text{IntraQ}_G(S, \mu)$ is the set of reachable states starting from $\{(s, \mu(s)) \mid s \in S\}$. We drop the subscript G in IntraQ when it is obvious from the context.

When the graph G is fixed, we can preprocess it to quickly answer subsequent queries. We now present three different algorithms for solving this query, where each of them trades off preprocessing time against time required to solve a query.

Alg1: The first algorithm is the standard way of solving such queries using no preprocessing. It is a saturation algorithm: each vertex v has a weight $l(v)$. Initially, $l(s) = \mu(s)$ for $s \in S$, and $\overline{0}$ for other vertices. Next, the rules $l(v) := l(v) \oplus (l(u) \otimes \lambda(u, v))$ for each edge (u, v) are used to update the weights until a fixpoint is reached. Then $l(v)$ is the required value for $\text{IntraQ}(S, \mu)(v)$. This requires time $O_s(|E|\mathcal{H})$, where $\mathcal{H}$ is the height of the weight domain, and the notation $O_s(.)$ denotes the asymptotic number of semiring operations. (Because we consider weights as black boxes, the algorithms in this paper seek to minimize the number of semiring operations.)

The disadvantage of this method, which our other algorithms address, is that it requires a fixpoint computation to be performed; this is reflected in the cost by the dependence on the height $\mathcal{H}$ of the weight domain, which can be large.

Alg2: The second algorithm does the obvious preprocessing. It precomputes the values $\text{MOP}[v_1, v_2]$ for each $v_1, v_2 \in V$ by solving $\text{IntraQ}(\{v_1\}, (v_1 \mapsto \overline{1}))$ for each v_1 using Alg1. Thus, preprocessing time is $O_s(|V||E|\mathcal{H})$. Once these

MOP values are available, IntraQ$(S, \mu)(v)$ can be solved from its definition in time $O_s(|S|)$. Thus, IntraQ(S, μ) can be solved in time $O_s(|S||V|)$, which is independent of $\mathcal{H}$. This may seem like the most efficient approach, but we show next that one can do better.

Consider the graph in Fig. 2. Suppose that $S = \{v_2, v_3\}$, $\mu(v_2) = w_2$, and $\mu(v_3) = w_3$. Then Alg2 would require approximately $2|V|$ semiring operations because it considers v_2 and v_3 separately from each other. However, notice that vertex v_4 *dominates* all other vertices with respect to v_2 and v_3, i.e., any path in the graph starting at v_2 or v_3 must pass through v_4 before reaching vertices v_5 to v_k (and vertex v_1 is unreachable). Based on this observation, we can prove that IntraQ$(S, \mu)(v_i)$ = IntraQ$(S, \mu)(v_4) \otimes$ MOP$[v_4, v_i]$ for $v_i \in \{v_5, \cdots, v_k\}$ Therefore, we only need to compute IntraQ$(S, \mu)(v_4)$ and other values can follow from this value using just one operation. This method would only require, approximately, $|V|$ number of operations.

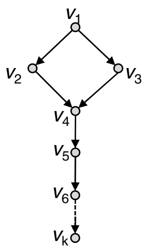


Fig. 2. A graph

This observation can be generalized to say that the weight on a vertex should be computed before the weights on vertices dominated by it are computed. This has been already captured nicely by Tarjan's algorithm [23] to solve path problems on graphs. However, it has only been used in the context of solving a single query, which we generalize to multiple queries. First, we summarize the essential details of Tarjan's algorithm.

Definition 7. *A* ***path expression*** *is a regular expression over the edges of a graph defined using the following grammar:* $r := \emptyset \mid \varepsilon \mid e \mid r_1.r_2 \mid r_1 \cup r_2 \mid r^*$ *where e is an edge in the graph. A path expression r is said to represent the set of paths in the language $\mathcal{L}(r)$ of r when interpreted as a regular expression. Furthermore, a path expression is said to be of type (u, v) if all paths in $\mathcal{L}(r)$ go from vertex u to vertex v.*

For example, for the graph in Fig. 2, the expression $((e_{12}.e_{24} \cup e_{13}.e_{34}).e_{45})$, where e_{ij} is the edge (v_i, v_j), denotes the set of all paths from v_1 to v_5, and is of type (v_1, v_5).

We extend λ to path expressions as follows: $\lambda(\emptyset) = \overline{0}, \lambda(\varepsilon) = \overline{1}, \lambda(r_1.r_2) = \lambda(r_1) \otimes \lambda(r_2)$, $\lambda(r_1 \cup r_2) = \lambda(r_1) \oplus \lambda(r_2)$, and $\lambda(r^*) = \lambda(r)^*$. Here, we define the weight w^* as the infinite combine $\overline{1} \oplus w \oplus (w \otimes w) \oplus (w \otimes w \otimes w) \oplus ...$, which exists because of Defn. 2(item 4). One can show that $w^* = (\overline{1} \oplus w)^{\mathcal{H}}$, and calculate it using repeated squaring in time $O_s(\log \mathcal{H})$. Consequently, the following lemma holds. (We define $|r|$ to be the length of the expression.)

Lemma 1. *For a path expression r and λ defined as above, $\lambda(r) = \bigoplus\{\lambda(\sigma) \mid \sigma \in \mathcal{L}(r)\}$. Moreover, it can be calculated in time $O_s(|r| \log \mathcal{H})$.*

Tarjan's algorithm is based on computing path expressions to represent the set of paths between each pair of vertices. However, instead of computing a separate path expression for each pair of vertices, it computes a *path sequence*, which is a more concise way of representing all paths in a graph.

Definition 8. *A* ***path sequence*** *of a directed graph $G = (V, E)$ is a sequence (r_1, u_1, v_1), $(r_2, u_2, v_2), \cdots (r_k, u_k, v_k)$, where $u_i, v_i \in V$, r_i is a path expression of type (u_i, v_i) such that for any nonempty path σ in G, there is a sequence of indices $1 \leq i_1 < i_2 < \cdots < i_l \leq k$ and a partition of σ into nonempty paths $\sigma = \sigma_1 \sigma_2 \cdots \sigma_l$ and $\sigma_j \in \mathcal{L}(r_{i_j})$ for all $1 \leq j \leq l$.*

Fig. 3(a) is an algorithm that uses a path sequence to create path expressions $r[s, v]$ that represent the set of all paths from s to v, for each $v \in V$ and a fixed $s \in V$ [23]. Using Lemma 1, we get $\mathrm{MOP}[s, v] = \lambda(r[s, v])$. Equivalently, the path expressions can be evaluated first and then put together to get the MOP weights, as shown in Fig. 3(b).

```
1: // initialize
2: for all $v \in V$ do
3:     $r[s, v] := \emptyset$
4: end for
5: $r[s, s] := \varepsilon$
6: // solve
7: for $i = 1$ to $k$ do
8:     $r[s, v_i] := r[s, v_i] \cup (r[s, u_i].r_i)$
9: end for
```
(a)

```
1: // initialize
2: for all $v \in V$ do
3:     $\mathrm{MOP}[s, v] := \overline{0}$
4: end for
5: $\mathrm{MOP}[s, s] := \overline{1}$
6: // solve
7: for $i = 1$ to $k$ do
8:     $\mathrm{MOP}[s, v_i] := \mathrm{MOP}[s, v_i] \oplus (\mathrm{MOP}[s, u_i] \otimes \lambda(r_i))$
9: end for
```
(b)

Fig. 3. Computing MOP values using the path sequence $\{(r_i, u_i, v_i)\}_{i=1}^{k}$

Tarjan's algorithm computes a path sequence for a graph in time $O(|E| \log |V| + \delta)$, where δ is a term that denotes the *irreducibility* factor of the graph. For reducible graphs, $\delta = 0$ and, in general, δ is bounded by $|V|^3$. Because the graphs we work with come from CFGs of procedures, they are mostly reducible and the δ term can be ignored (which is confirmed by our experiments). Evaluating all path expressions takes time $O_s((|E| \log |V| + \delta) \log \mathcal{H})$. After that, given a vertex s, solving for $\mathrm{MOP}[s, v]$ for all vertices v requires time $O_s(|E| \log |V| + \delta)$, which is almost linear in the size of the graph. We ignore δ in the rest of the paper.

Alg3: Suppose that we wish to solve $\textsc{IntraQ}(S, \mu)$ on $G = (V, E, \lambda)$. We construct a new graph $G' = (V', E', \lambda')$ by adding a new vertex to G: for some $v_0 \notin V$, $V' = V \cup \{v_0\}$, $E' = E \cup \{(v_0, s) \mid s \in S\}$, $\lambda' = \lambda \cup \{[(v_0, s) \mapsto \mu(s)] \mid s \in S\}$. Then $\mathrm{MOP}_{G'}[v_0, v] = \textsc{IntraQ}_G(S, \mu)(v)$. Thus, we need to compute MOP values on G'. This trick is similar to the standard one of reducing a multi-source reachability problem to a single-source reducibility problem. The following observation shows that a path sequence for G' can be computed from that of G:

Lemma 2. *If ps is a path sequence of G, then by concatenating the sequence $\{(v_0, s, (v_0, s)) \mid s \in S\}$ (with any arbitrary order chosen to enumerate vertices in S) in front of ps, one obtains a path sequence for G'.*

The preprocessing step of $\textsc{Alg3}$ computes a path sequence for G and evaluates the weight of each of its path expressions. Then to solve each query, $\textsc{Alg3}$ uses the path sequence for G', constructed using Lemma 2, as input to the algorithm in Fig. 3(b). This gives us the required weights $\textsc{IntraQ}_G(S, \mu)(v)$ as $\mathrm{MOP}_{G'}[v_0, v]$. $\textsc{Alg3}$ requires $O_s(|E| \log |V| \log \mathcal{H})$ preprocessing time and

$O_s(|S| + |E| \log |V|)$ time to solve each query. This is much better than ALG2 because for CFGs, $|E|$ is usually $O(|V|)$. We used ALG3 in our experiments.

4 Solving Multiple Queries on WPDSs

For graphs, the number of vertices is finite, but for WPDSs, the number of configurations may be infinite (when the program is recursive), or very large (exponential in the number of procedures when not recursive). For this reason, sets of weighted configurations (or program states) are represented symbolically using weighted automata [18].

Definition 9. *A weighted automaton $\mathcal{A}$ is a finite-state automaton where each transition is additionally labeled with a weight. The weight of a path in the automaton is obtained by taking an extend of the weights on the transitions in the path in the backward direction. The automaton is said to accept a configuration $\langle p, u \rangle$ with weight w, denoted by $\mathcal{A}(\langle p, u \rangle)$, if w is the combine of weights of all accepting paths for u starting from state p in $\mathcal{A}$ ($w = \overline{0}$ if u is not accepted). The set of states of $\mathcal{A}$ is assumed to contain P, the set of PDS states.*

A weighted automaton $\mathcal{A}$ represents the set of states $\mathcal{R}(\mathcal{A}) = \{(c, \mathcal{A}(c)) \mid \mathcal{A}(c) \neq \overline{0}\}$. An important result is that the set *poststar*$(\mathcal{R}(\mathcal{A}))$ (as defined in §2) can also be represented by a weighted automaton [18]. For brevity, we call such an automaton *poststar*$(\mathcal{A})$. Our goal is to preprocess a given WPDS so that *poststar*$(\mathcal{A})$ can be computed quickly for any given $\mathcal{A}$.

An example is shown in Fig. 4(a). Note how the weight for a configuration $\langle p, n_7\ n_3 \rangle$ is represented in a compositional way in the automaton. Procedure `bar` is analyzed independently of its calling context, resulting in weight w_4 for transition (p, n_7, q). The weight w_2 at the call site n_3 to `bar` is captured on the transition (q, n_3, acc), resulting in a total weight of $w_4 \otimes w_2$ for $\langle p, n_7\ n_3 \rangle$. We will use the fact that procedures are analyzed independently of their calling context (also customary in most summary-based interprocedural analyses) in our favor. (As we shall see later, this implies that weights have to be propagated from a procedure to its callers, but not to procedures that it calls.)

Fix $\mathcal{W} = ((P, \Gamma, \Delta), \mathcal{S}, f)$ to be a WPDS. Fix $\mathcal{A}_{\text{start}}$ to be the input query automaton, for which we want to compute *poststar*$(\mathcal{A}_{\text{start}})$. To simplify the discussion, assume that $\mathcal{W}$ was created from a program as described in §2, and $P = \{p\}$ is a singleton set (our implementation handles any WPDS, however).

Preprocessing. (i) First, we compute a summary for each procedure. (For a procedure starting at node e, it is defined as IMOP$(\langle p, e \rangle, \langle p, \varepsilon \rangle)$). Using these summaries, we construct a weighted graph for each procedure from its CFG: the call edges (from call site to return site) are replaced with a summary of the called procedure. For $\gamma \in \Gamma$, let PR$_\gamma$ be the procedure that contains γ, G_γ be the weighted graph for PR$_\gamma$, e_γ its unique entry node, and x_γ its unique exit node. (Note that MOP$_{G_\gamma}[e_\gamma, x_\gamma]$ also equals the summary for PR$_\gamma$.) Next, for each weighted graph G of a procedure, we compute: (ii) its path sequence

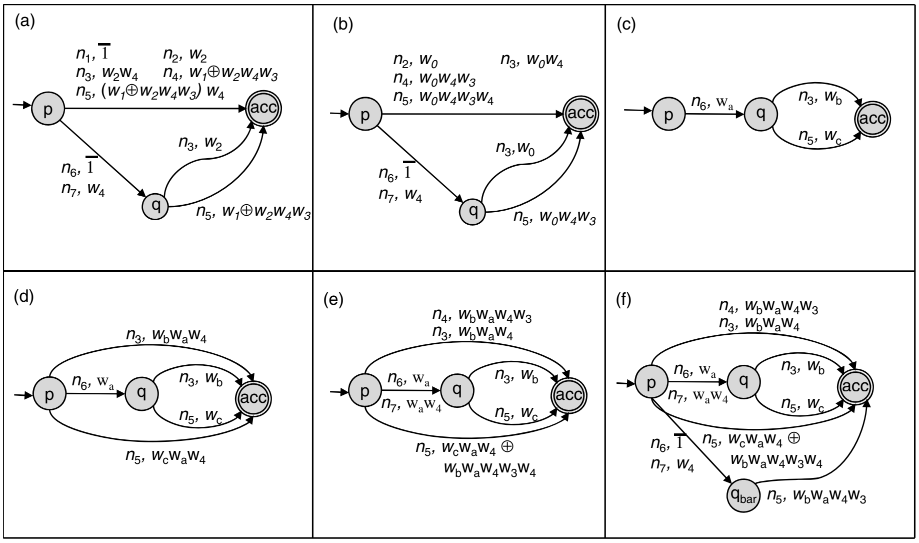


Fig. 4. Various automata related to the WPDS of Fig. 1. In all the weighted automata, juxtaposition of weights denotes their extend, *acc* is the accepting state, and parallel transitions have sometimes been collapsed into a single edge. Labels on transitions are (stack symbol, weight) pairs. (a) An automaton for $\mathit{poststar}(\{(\langle p, n_1\rangle, \overline{1})\})$. (b) An automaton for $\mathit{poststar}(\{(\langle p, n_2\rangle, w_0)\})$. (c) Automaton $\mathcal{A}_{\mathrm{start}}$. (d) Automaton $\mathcal{A}_{\mathrm{pop}}$ obtained after running the pop-phase.(e) Automaton $\mathcal{A}_{\mathrm{int}}$ created while running the growth phase on $\mathcal{A}_{\mathrm{pop}}$. (f) The final result of running the growth phase on $\mathcal{A}_{\mathrm{pop}}$.

(preprocessing for Alg3) and (iii) values $\mathrm{MOP}_G[\gamma, x_\gamma]$ and $\mathrm{MOP}_G[e_\gamma, \gamma]$ for each node γ of the procedure.

The procedure summaries can be computed using standard algorithms, after which the path sequences can be constructed using Tarjan's algorithm. This would be an acceptable solution, but we can do better. We use our techniques from [9] to compute both of these at the same time. Briefly, the call-return edge in the CFG of a procedure is labeled with a variable whose value stands for the (as yet uncomputed) summary of the called procedure. Then the procedure summary is represented using a path expression (from entry node to return node) computed from its path sequence. This expression will have variables standing for summaries of called procedures. This gives rise to a set of equations whose solution solves for all summaries. In [9], we showed that this technique provides up to 5 times speedup over standard algorithms for computing procedure summaries, and we obtain a path sequence as a by-product. The path sequences can then be used to quickly compute the required MOP values for (iii) [23].

ICFG-version. Before describing how our algorithm works with weighted automata, we briefly describe how it would work with ICFGs (after preprocessing). (Full details are in [10].) Suppose that we are given a set R of node-weight pairs

$$\begin{array}{rlr}
\langle p, \gamma_1\ \gamma_2\ \gamma_3 \cdots \gamma_n \rangle & \Rightarrow^{\sigma_1} & \langle p, \gamma_2\ \gamma_3 \cdots \gamma_{k+1}\ \gamma_{k+2} \cdots \gamma_n \rangle \\
& \Rightarrow^{\sigma_2} & \langle p, \gamma_3 \cdots \gamma_{k+1}\ \gamma_{k+2} \cdots \gamma_n \rangle \\
& \Rightarrow^{*} & \cdots \\
& \Rightarrow^{\sigma_k} & \langle p, \gamma_{k+1}\ \gamma_{k+2} \cdots \gamma_n \rangle \\
& \Rightarrow^{\sigma_{k+1}} & \langle p, u_1\ u_2 \cdots u_j\ \gamma_{k+2} \cdots \gamma_n \rangle
\end{array}$$

Fig. 5. A path in the PDS's transition relation; $u_i \in \Gamma, j \geq 1, k \leq n, \sigma_h \in \Delta^*$

(starting states), where the nodes may be from multiple procedures, and we want to calculate the reachable set of node-weight pairs.

One challenge is to isolate the intraprocedural work. An IntraQ query on a set $S = \{s_1, \cdots, s_k\}$ can also be solved by making a separate query for each s_i and taking a combine of the results, but this is far less efficient than making a single query on S. Thus, we want to minimize the number of IntraQ queries made for each procedure. For example, for the program in Fig. 1, suppose $R = \{(n_6, w_a), (n_4, w_b)\}$. Then the pair (n_6, w_a) can produce the pairs $(n_3, w_a \otimes w_4)$ and $(n_5, w_a \otimes w_4)$ inside `foo`, when `bar` returns. We would then like to make just one IntraQ query on `foo` with $S = \{n_3, n_4, n_5\}$ (and appropriate weights), instead of making a query with just n_4 first, and then realizing that the procedure has to be explored again from n_3 and n_5.

The algorithm proceeds in two phases. The first phase moves across procedure boundaries: if $(n, w) \in R$ then we propagate this weight to the callers of Pr_n. We add $(r, w \otimes \text{MOP}_{G_n}[n, x_n])$ to R for each return site r of calls to Pr_n (if the pair (r, w') was already present in R, then change w' to $w' \oplus (w \otimes \text{MOP}_{G_n}[n, x_n])$). This continues until saturation. The use of (precomputed) $\text{MOP}[n, x_n]$ weights allow us to quickly jump from a procedure to its callers.

The second phase is intraprocedural. If $(n_1, w_1), \cdots, (n_k, w_k) \in R$ and the n_i are from the same procedure, run IntraQ$(\{n_1, \cdots, n_k\}, [n_i \mapsto w_i])$ to get weights on all other nodes in the procedure. This is repeated for all procedures. The resulting node-weight pairs represent all reachable states.

The extension of these ideas to WPDSs have two complications: First, configurations add constraints on how weights get propagated to callers. For example, starting at configuration $\langle p, \gamma_1\gamma_2 \rangle$ constrains weight propagation to γ_2 when Pr_{γ_1} returns (and not to its other return sites). Second, the number of configurations may be infinite, forcing us to use automata-based symbolic representations.

The above ICFG version only required at most one IntraQ query per procedure, which is ideal. The general version for WPDSs requires slightly more queries: at most $|Q|$ queries per procedure, where Q is the set of states of $\mathcal{A}_{\text{start}}$.

WPDS-version. Consider a path $\sigma \in \Delta^*$ in the transition relation of a PDS that starts from a configuration $\langle p, \gamma_1\gamma_2 \cdots \gamma_n \rangle$. It can always be decomposed as $\sigma = \sigma_1\sigma_2 \cdots \sigma_k\sigma_{k+1}$ (see Fig. 5), such that $\langle p, \gamma_i \rangle \Rightarrow^{\sigma_i} \langle p, \varepsilon \rangle$ for $1 \leq i \leq k$ and $\langle p, \gamma_{k+1} \rangle \Rightarrow^{\sigma_{k+1}} \langle p, u_1u_2 \cdots u_j \rangle$ (or σ_{k+1} is empty when $k = n$). In other words, σ_i, $i \leq k$ is the rule sequence whose net effect is to pop off γ_i without looking at the stack below it, and σ_{k+1} is the rule sequence that does not look below γ_{k+1} but can replace it and add more symbols on top of the stack. We call the part where symbols are popped $(\sigma_1, \cdots, \sigma_k)$ the *pop phase* and the part where the stack grows (σ_{k+1}), the *growth phase*.

This property holds because PDS rules can only look at the top of the stack. For σ to touch γ_2, it must first pop off γ_1. When it does pop it off, this prefix would be σ_1 (and repeat inductively). If it does not pop off γ_1, then σ is already in the growth phase.

We compute *poststar*$(\mathcal{A}_{\text{start}})$ by separating these two phases: First, we compute $\mathcal{A}_{\text{pop}}$ that accepts all configurations reachable after the pop phase. Next, we start from $\mathcal{A}_{\text{pop}}$ and build $\mathcal{A}_{\text{final}}$ that accepts all configurations reachable after the growth phase. The former part is similar to the first phase of the ICFG-version of the algorithm, and the latter will correspond to the intraprocedural part (similar to the second phase of the ICFG-version). A running example is shown in Fig. 4$(c) - (f)$.

Terminology. (i) A transition t with weight w is *added* to a weighted automaton $\mathcal{A}$ as follows: if t does not exist in $\mathcal{A}$, then insert it with weight w. If it exists in $\mathcal{A}$ with weight w', then change its weight to $w' \oplus w$. (ii) We say that $\mathcal{A}$ accepts a configuration c with weight *at least* w if $\mathcal{A}(c) \sqsubseteq w$ (Defn. 2, item 4). Note that all configurations are accepted with weight at least $\overline{0}$. (iii) The pop and growth phases are *saturation procedures*. They convert input $\mathcal{A}$ to output $\mathcal{A}'$ by adding transitions to $\mathcal{A}$ until a fixpoint is reached; the fixpoint is the desired output $\mathcal{A}'$. Consequently, for all c, $\mathcal{A}'(c) \sqsubseteq \mathcal{A}(c)$, and thus for all c, $\mathcal{A}_{\text{final}}(c) \sqsubseteq \mathcal{A}_{\text{pop}}(c) \sqsubseteq \mathcal{A}_{\text{start}}(c)$.

Pop Phase. Let w_γ be the weight with which γ can be popped, i.e., $w_\gamma = \text{IMOP}(\langle p, \gamma \rangle, \langle p, \varepsilon \rangle) = \text{MOP}_{G_\gamma}[\gamma, x_\gamma]$, which has been precomputed. We perform saturation on $\mathcal{A}_{\text{start}}$: if it accepts a configuration $\langle p, \gamma\ \gamma'\ u \rangle$, for any $u \in \Gamma^*$, with weight w, we make it accept $\langle p, \gamma'\ u \rangle$ with weight at least $w \otimes w_\gamma$, and repeat until a fixpoint is reached. This is done as follows: if (p, γ, q_1) and (q_1, γ', q_2) are transitions in the automaton with weight w_1 and w_2, respectively, then add the transition (p, γ', q_2) with weight $w_2 \otimes w_1 \otimes w_\gamma$ to the automaton. This process terminates because the number of new transitions added is bounded by $|T|$, where T is the set of transitions of $\mathcal{A}_{\text{start}}$. (This is because a transition (q_1, γ, q_2) in $\mathcal{A}_{\text{start}}$ can cause at most a single transition (p, γ, q_2) to be added to $\mathcal{A}_{\text{pop}}$.) Defn. 2 (item 4) ensures that weights on them can change at most $\mathcal{H}$ times. Moreover, the running time is bounded by $O_s(|T|\mathcal{H})$. Fig. 4(d) shows an example: $\mathcal{A}_{\text{start}}$ accepts $\langle p, n_6 n_3 \rangle$ with weight $w_b \otimes w_a$, the weight with which n_6 can be popped is w_4, and $\mathcal{A}_{\text{pop}}$ accepts $\langle p, n_3 \rangle$ with weight $w_b \otimes w_a \otimes w_4$.

Growth Phase. For the growth phase, we need to consider all configurations reachable from the top symbols of currently accepted configurations, i.e., if $\langle p, \gamma\ u \rangle$, $u \in \Gamma^*$, is accepted by $\mathcal{A}_{\text{pop}}$ with weight w, and $\langle p, \gamma \rangle \Rightarrow^* \langle p, u' \rangle, u' \in \Gamma^+$ then $\langle p, u'\ u \rangle$ should be accepted by $\mathcal{A}_{\text{final}}$ with weight at least $w \otimes \text{IMOP}(\langle p, \gamma \rangle, \langle p, u' \rangle)$. Now we make use of the observation that called procedures are analyzed independently of their calling context, and reduce this phase to an intraprocedural one. For instance, see Fig. 4(b)—the weight w_0 need not be propagated to transitions involving nodes from `bar`.

The growth phase proceeds in two parts. The first part constructs automaton $\mathcal{A}_{\text{int}}$ such that if $\mathcal{A}_{\text{pop}}$ accepted configuration $\langle p, \gamma\ u\rangle$ with weight w and $\langle p,\gamma\rangle \Rightarrow^* \langle p,\gamma'\rangle$ then $\mathcal{A}_{\text{int}}$ accepts $\langle p, \gamma'\ u\rangle$ with weight at least $w \otimes \text{IMOP}(\langle p,\gamma\rangle, \langle p,\gamma'\rangle)$. This part requires running IntraQ queries.

For the first part, note that if $\langle p,\gamma\rangle \Rightarrow^* \langle p,\gamma'\rangle$, then γ' must be from the same procedure as γ (otherwise, the stack length would be different). Then $\text{IMOP}(\langle p,\gamma\rangle, \langle p,\gamma'\rangle) = \text{MOP}_{G_\gamma}[\gamma,\gamma']$. Hence, it suffices to do the following: if (p,γ,q) is a transition with weight w in $\mathcal{A}_{\text{pop}}$ then add transitions (p,γ',q) to it, for each γ' in the same procedure as γ, with weight $w \otimes \text{MOP}_{G_\gamma}[\gamma,\gamma']$. This may add transitions with weight $\overline{0}$ if γ' is not reachable from γ, but such transitions can be removed without changing the meaning of a weighted automaton.

The above process can be optimized. Instead of looking at each transition in isolation, we handle them in bulk. For a state q of $\mathcal{A}_{\text{pop}}$, and a procedure Pr, let S_q^{Pr} be the set of nodes s in Pr such that (p,s,q) is a transition in $\mathcal{A}_{\text{pop}}$. Let μ_q^{Pr} be such that $\mu_q^{\text{Pr}}(s)$ is the weight on (p,s,q). Then add transition (p,s',q) with weight $\text{IntraQ}(S_q^{\text{Pr}}, \mu_q^{\text{Pr}})(s')$. It is easy to see that this imitates the above process, but is more efficient. This results in automaton $\mathcal{A}_{\text{int}}$. The running time is bounded by that required to answer $|Q||\text{Proc}|$ number of IntraQ queries, where Q is the set of states of $\mathcal{A}_{\text{pop}}$ (same as those of $\mathcal{A}_{\text{start}}$), and $|\text{Proc}|$ is the number of procedures in the program.

An example is shown in Fig. 4(e): the algorithm invokes $\text{IntraQ}_{\texttt{bar}}(\{n_6\}, [n_6 \mapsto w_a])$ to add transitions between p and q. Next, it invokes $\text{IntraQ}_{\texttt{foo}}(\{n_3, n_5\}, [n_3 \mapsto w_b \otimes w_a \otimes w_4, n_5 \mapsto w_c \otimes w_a \otimes w_4])$ to add transitions between p and acc.

The second part of the growth phase adds transitions to accept configurations of called procedures. For each procedure Pr, add a new state q_{Pr} to $\mathcal{A}_{\text{int}}$, and let $\text{Called}(\text{Pr})$ be *false* initially. Now repeat the following: if (p,γ,q) is a transition with weight w_1 and $\langle p,\gamma\rangle \hookrightarrow \langle p, c\ r\rangle$ is a WPDS rule with weight w_2, then (i) if $\text{Called}(\text{Pr}_c)$ is *false*, then set it to *true* and add transitions $(p,\gamma',q_{\text{Pr}_c})$ with weight $\text{MOP}_{\text{Pr}_c}[c,\gamma']$, for each node γ' in Pr_c; (ii) add transition (q_{Pr_c}, r, q) with weight $w_1 \otimes w_2$.

The intuition here is that with $\sigma = \langle p,\gamma\rangle \hookrightarrow \langle p, c\ r\rangle$, $\langle p, \gamma\ u\rangle \Rightarrow^\sigma \langle p, c\ r\ u\rangle$ for any $u \in \Gamma^*$. Addition of transitions (p, c, q_{Pr_c}) and (q_{Pr_c}, r, q) ensures that the latter configuration is accepted (with appropriate weights). Next, c can reach node γ' in the same procedure with weight $\text{MOP}_{\text{Pr}_c}[c,\gamma']$, for which the transitions $(p,\gamma',q_{\text{Pr}_c})$ are added. Note that the weight at the call site $(w_1 \otimes w_2)$ gets stored on the transition (q_{Pr_c}, r, q). Thus, Pr_c is analyzed independently of this weight and the weights on transitions $(p,\gamma',q_{\text{Pr}_c})$, for each γ' in Pr_c, are independent of the input query.

This process terminates because only a finite number of states are added. The trick of bounding the number of states is common in reachability algorithms for PDSs [18,21]. The running time is bounded by $O_s(|Ret||Q|) + O(|\Gamma|)$, where Ret

is the set of return sites in the program. This running time is subsumed by that of the first part. Fig. 4(f) shows an example.

Complexity. First, we discuss the complexity of solving a query after preprocessing has been completed. Let Q be the set of states of $\mathcal{A}_{\text{start}}$, and T the set of its transitions. The pop phase has running time $O_s(|T|\mathcal{H})$. The growth phase, when using ALG3 for INTRAQ queries, has running time $O_s(|Q||\mathit{Proc}||E|\log|V|)$, where $|\mathit{Proc}|$ is the number of procedures in the program, and $|E|$ and $|V|$ are the average number of nodes per procedure. This gives a total worst-case running time of $O_s(|T|\mathcal{H}+|Q||\mathit{Proc}||E|\log|V|)$. The number of nodes per procedure usually remains constant even as program size increases. Treating $\log|V|$ as a constant, and writing $|E||\mathit{Proc}|$ as $|\Delta|$ (the number of WPDS rules), we get a total complexity of $O_s(|T|\mathcal{H}+|Q||\Delta|)$. This is asymptotically better than the complexity of previous algorithms [18,9], which is $O_s(|T|\mathcal{H}+(|Q|+|\mathit{Proc}|)|\Delta|\mathcal{H})$ in each case. Note the reduced dependence on $\mathcal{H}$ for our algorithm (hence less fixpoint computation around loops and recursion).

If the initial set of configurations is finite (i.e., automaton $\mathcal{A}_{\text{start}}$ does not have any cycles), the running time of the pop phase can be bounded by $O_s(|T|)$, resulting in a total running time (after preprocessing) that is completely independent of the height of the weight domain (which is not true for any other WPDS reachability algorithm).

The complexity for preprocessing is dominated by the step that computes procedure summaries (and path sequences as a by-product). This just requires a single dataflow query, whose complexity is $O_s(|\mathit{Proc}||\Delta|\mathcal{H})$ [9]. Using path sequences to compute the other preprocessing information is fairly quick.

5 Experiments

We refer to the implementation of the algorithm in this paper as SWPDS (Summary-WPDS). We compare against saturation-based [18] and optimized [9] approaches for solving WPDS queries, of which we pick the better running time and refer to it as OWPDS (Old-WPDS).

We carried out experiments on WPDSs obtained from three different applications. The first application is affine-relation analysis (ARA) of x86 programs [1]. A WPDS is produced from the x86 program using the weight domain for ARA described in [12]. The goal is to discover affine relationships (linear equalities) between machine registers.

The first experiment is to find out loop invariants (where loops are discovered by Bourdoncle's decomposition technique [3]). For outermost loops in a procedure, a loop summary is obtained as the weight IMOP$(\langle p, \texttt{head}\rangle, \langle p, \texttt{head}\rangle)$, where `head` is the head of the loop. This can be calculated by computing $\mathcal{A} = \mathit{poststar}(\{(\langle p, \texttt{head}\rangle, \overline{1})\})$, and $\mathcal{A}(\langle p, \texttt{head}\rangle)$. Loop invariants can be calculated easily from these summaries. These invariants give the conditions that hold at the head of the loop and are re-established after each iteration of the loop. We use common Windows executables, including code for the called

Table 1. ARA experiments. The speedup is reported for SWPDS versus OWPDS2.

				Time (s)					Constant Registers				Other Invariants			
Prog	Insts	Procs	Loops	OWPDS	OWPDS2	Setup	SWPDS	Speedup	0	1-3	4-6	7-8	0	1	2	≥ 3
latex	63711	609	280	168	5.7	28	4.3	**1.3**	53	44	152	31	124	125	31	0
attrib	103473	964	537	290	8.3	46	5.1	**1.7**	97	114	271	55	227	254	56	0
ftp	130352	1271	634	731	13.5	26	8.7	**1.6**	130	126	320	58	290	280	64	0
notepad	167430	1609	749	597	12.1	43	8.2	**1.5**	162	156	369	62	325	336	87	1
cmd	192579	1783	869	3415	24.1	64	18.0	**1.3**	256	156	391	66	431	355	80	3

Table 2. Experiments on Boolean programs: (*i*) Forward and backward reachability from the set of configurations $n\ Ret^*$. The node n was chosen randomly, and running times, reported in seconds, were averaged across 5 queries. The speedup is reported per query, ignoring the setup time. (*ii*) Simulated CBMC queries. The speedup reported takes the setup time into account.

				Forward Reach.			Backward Reach.			CBMC		
Prog	Nodes	Procs	Setup	SWPDS	OWPDS	Speedup	SWPDS	OWPDS	Speedup	SWPDS	OWPDS	Speedup
bugs5	36971	291	11.9	4.2	18.4	**4.4**	1.4	8.4	**5.8**	98	183	**1.7**
unified-serial	38234	291	15.4	5.3	24.5	**4.7**	1.7	12.1	**7.1**	129	238	**1.6**
slam	7161	97	3.5	2.7	14.2	**5.3**	0.4	2.1	**6.0**	87	115	**1.3**
iscsiprt1	4803	82	0.6	0.27	0.84	**3.1**	0.06	0.36	**6.0**	6.3	12	**1.7**
ufloppy13	5679	64	1.5	0.6	2.0	**3.1**	0.07	0.7	**10.3**	12	20	**1.5**

libraries, and ran the experiments on a 3.2 GHz P4 processor with 3.3GB RAM running Windows XP.

A conventional way to solve these queries would be to compute the procedure summaries, plug them at the call-sites and then solve each loop as an intraprocedural problem. We call this technique OWPDS2. It uses ALG1 to solve each loop. Tab. 1 reports the following timings: the time taken to answer each query independently (OWPDS); the time taken by OWPDS2 (after procedure summaries have been computed); the preprocessing time for SWPDS (Setup); and the time taken to answer all queries using SWPDS, after preprocessing. We make two comparisons: (SWPDS+Setup) versus OWPDS, for which we are about 17 times faster (not shown in the table), and SWPDS versus OWPDS2, for which we are 1.5 times faster. We do not take the setup times into account in the second comparison because SWPDS preprocessing only computes procedure summaries (other preprocessing is unnecessary for this application).

Tab. 1 also shows a distribution of the obtained loop invariants. Loop invariants that indicate that a register remains unchanged after each loop iteration (even though it may get modified inside the loop) are reported separately from other kinds of invariants. The last eight columns show the number of loops that have a certain number of constant registers or affine invariants. For example, in `latex`, 53 loops do not have any constant registers, and 31 loops have 2 linearly independent invariants. These invariants can be beneficial to other analysis (e.g., they identify loop-induction variables).

The second application is Boolean program verification using MOPED [21]. Boolean programs are converted to WPDSs, and dataflow analysis is used for proving properties of the program. In our experiments, the Boolean programs were obtained as a result of predicate abstraction. The following experiments were run on a 3GHz P4 processor with 2GB RAM running Linux, and the results are reported in Tab. 2.

We ran queries starting from the set of configurations $(n\mathit{Ret}^*)$, where n is a program node and *Ret* is the set of return-site nodes. Such queries are useful for finding out the net effect in going from one program statement to another (for finding dependencies between the two). After the setup time, SWPDS was 4 times than OWPDS on forward reachability queries, and 7 times faster on backward reachability (our algorithm for solving backward reachability is given in [10]). This experiment also shows that two dataflow queries are enough to recover the SWPDS preprocessing time.

The third application (also based on Moped, running on the same Linux platform), considers context-bounded model checking (CBMC) [13], which aims to find all reachable states of a concurrent program under a bound on the number of context switches. Because we lack a front-end to abstract concurrent programs, we performed simulated experiments on sequential programs. We assume that the global variables of the program are shared with an environment that can randomly change their value, and the environment itself does not possess any local state. We ran one branch of CBMC along which control is transferred to the environment 5 times. Essentially, this requires the following: for random global states $g_1, \cdots, g_5$, if $\mathcal{A}_0$ describes the initial configuration of the (sequential) program, then compute $\mathcal{A}_1 = \mathit{poststar}(\mathcal{A}_0)$ and $\mathcal{A}_{i+1} = \mathit{poststar}(\textsc{Modify}(\mathcal{A}_i, g_i))$ for $i = 1$ to 5. Here, $\textsc{Modify}(\mathcal{A}, g)$ is an automaton that represents the same set of states as $\mathcal{A}$, but with the shared state changed to g. Because the result of running *poststar* is a bigger automaton than the original one, we report the total time taken to run all the queries. The average speedup was 1.6 times.

The varying amount of speedups in the different applications seems to be related to the size of S in $\textsc{IntraQ}(S, \mu)$ queries. For the ARA experiments, $|S|$ was always 1 (maximum speedup over OWPDS); for the second application, S was usually the set of return sites in a procedure; for CBMC, S consisted of all nodes in a procedure (least amount of speedup). Worst-case complexity does not predict this effect; this is an observation about measured behavior.

6 Related Work

The goal of incremental program analysis [4,14,11,20,5] is to reuse as much information as possible from previous fixpoint computations to calculate a new fixpoint when a small change is made to the program. Our work has aspects that resemble incremental computing in that we avoid recomputing the same information in response to changes in the query. However, we have a single preprocessing step to compute summaries and path sequences; this information is used by multiple dataflow queries, but there is no additional information tabulated during one query for use by a later query. This is because we do not look into the weights (and avoid caching computations over them), and base our optimizations only on the program control structure.

Another closely related category of work is that on demand-driven dataflow analysis [6,15,19]. There the focus is to do only as much work as is required to solve a query, and not redo it in a subsequent query. However, these techniques assume a particular form for the weights, and do look inside them (to work with *exploded CFGs*). We make fewer assumptions about the weights. These techniques would not be applicable to the weight domain we considered in our first application or be able to work with BDDs, as required by the other applications.

Technically, the most closely related piece of work is our previous work on speeding up a *single* dataflow query [9] called FWPDS. It used Tarjan's algorithm at the intraprocedural level to compute regular expressions for solving MOP values, and combined it with techniques like incremental computation of regular expressions to extend it for interprocedural analysis. In this paper, we use Tarjan's algorithm to compute path sequences. We combine it with a new WPDS reachability algorithm that shows how to summarize and reuse information at the interprocedural level. Moreover, FWPDS required the starting set of configurations to be in hand before it built the graphs on which it ran Tarjan's algorithm and may build different graphs for different queries, preventing it from sharing information between them. SWPDS outperforms FWPDS (included as OWPDS in §5).

References

1. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: Duesterwald, E. (ed.) CC 2004. LNCS, vol. 2985, Springer, Heidelberg (2004)
2. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: POPL (2003)
3. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: FMPA (1993)
4. Cai, J., Paige, R.: Program derivation by fixed point computation. SCP 11(3) (1989)
5. Conway, C.L., Namjoshi, K.S., Dams, D., Edwards, S.A.: Incremental algorithms for inter-procedural analysis of safety properties. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, Springer, Heidelberg (2005)
6. Duesterwald, E., Gupta, R., Soffa, M.L.: Demand-driven computation of interprocedural data flow. In: POPL (1995)
7. Graham, S., Wegman, M.: A fast and usually linear algorithm for global flow analysis. J. ACM 23(1) (1976)
8. Knoop, J., Steffen, B.: The interprocedural coincidence theorem. In: Pfahler, P., Kastens, U. (eds.) CC 1992. LNCS, vol. 641, Springer, Heidelberg (1992)
9. Lal, A., Reps, T.: Improving pushdown system model checking. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, Springer, Heidelberg (2006)
10. Lal, A., Reps, T.: Solving multiple dataflow queries using WPDSs. Technical Report 1632, University of Wisconsin-Madison (March 2008)
11. Liu, Y.A., Stoller, S.D., Teitelbaum, T.: Static caching for incremental computation. TOPLAS 20(3) (1998)
12. Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, Springer, Heidelberg (2005)

13. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, Springer, Heidelberg (2005)
14. Ramalingam, G., Reps, T.W.: A categorized bibliography on incremental computation. In: POPL (1993)
15. Reps, T.: Solving demand versions of interprocedural analysis problems. In: Fritzson, P.A. (ed.) CC 1994. LNCS, vol. 786, Springer, Heidelberg (1994)
16. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL (1995)
17. Reps, T., Lal, A., Kidd, N.: Program analysis using weighted pushdown systems. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, Springer, Heidelberg (2007)
18. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. In: SCP, vol. 58 (2005)
19. Sagiv, S., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theor. Comput. Sci. 167(1&2) (1996)
20. Saha, D., Ramakrishnan, C.R.: Incremental evaluation of tabled logic programs. In: Palamidessi, C. (ed.) ICLP 2003. LNCS, vol. 2916, Springer, Heidelberg (2003)
21. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, Technical Univ. of Munich, Munich, Germany (July 2002)
22. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis. In: Program Flow Analysis: Theory and Applications, Prentice-Hall, Englewood Cliffs (1981)
23. Tarjan, R.E.: Fast algorithms for solving path problems. J. ACM 28(3), 594–614 (1981)

Field Flow Sensitive Pointer and Escape Analysis for Java Using Heap Array SSA

Prakash Prabhu and Priti Shankar

Department of Computer Science and Automation,
Indian Institute of Science,
Bangalore 560012, India

Abstract. Context sensitive pointer analyses based on Whaley and Lam's *bddbddb* system have been shown to scale to large Java programs. We provide a technique to incorporate flow sensitivity for Java fields into one such analysis and obtain an escape analysis based on it. First, we express an intraprocedural *field flow sensitive analysis*, using Fink et al.'s Heap Array SSA form in Datalog. We then extend this analysis interprocedurally by introducing two new ϕ functions for Heap Array SSA Form and adding deduction rules corresponding to them. Adding a few more rules gives us an escape analysis. We describe two types of field flow sensitivity: *partial* (PFFS) and *full* (FFFS), the former without strong updates to fields and the latter with strong updates. We compare these analyses with two different (field flow insensitive) versions of Whaley-Lam analysis: one of which is flow sensitive for locals (FS) and the other, flow insensitive for locals (FIS). We have implemented this analysis on the *bddbddb* system while using the *SOOT* open source framework as a front end. We have run our analysis on a set of 15 Java programs. Our experimental results show that the time taken by our field flow sensitive analyses is comparable to that of the field flow insensitive versions while doing much better in some cases. Our PFFS analysis achieves average reductions of about 23% and 30% in the size of the points-to sets at load and store statements respectively and discovers 71% more "caller-captured" objects than FIS.

1 Introduction

A pointer analysis attempts to statically determine whether two variables may point to the same storage location at runtime. Many compiler optimizations like loop invariant code motion, parallelization and so on require precise pointer information in order to be effective. A precise pointer analysis can also obviate the need for a separate escape analysis. Of the various aspects of a pointer analysis for Java that affect its precision and scalability, two important ones are context sensitivity and flow sensitivity. A context sensitive analysis does not allow information from multiple calling contexts to interfere with each other. Although in the worst case, it can lead to exponential analysis times, context sensitivity is important, especially in case of programs with short and frequently invoked

© Springer-Verlag Berlin Heidelberg 2008

methods. A flow sensitive analysis takes control flow into account while determining points to relations at various program points. An analysis could be flow sensitive for just scalars or for object fields too. Since it can avoid generation of spurious points to relations via non-existent control flow paths, flow sensitivity is important for precision of a pointer analysis. One of the most scalable context sensitive pointer analysis for Java is due to Whaley and Lam [1], based on the *bddbddb* system. However, it is flow sensitive just for locals and not for object fields. The analysis of Fink et al. [2], based on the Heap Array SSA form [2], is flow sensitive for both locals and fields. However, it is intraprocedural and context insensitive. In this work, we develop an analysis similar to that of Fink et al., extend it interprocedurally and integrate it into the context sensitive framework of Whaley and Lam. The contributions of this paper can be summarized as follows:

– We formulate two variants of a field flow sensitive analysis using the Heap Array SSA Form in Datalog: *partial field flow sensitive analysis* (PFFS) and *full field flow sensitive analysis* (FFFS). Section 2 gives an overview of the Heap Array SSA form and describes our formulation of intraprocedural field flow sensitive analysis in Datalog.
– We extend the Heap Array SSA form interprocedurally by introduction of two new ϕ functions: the *invocation* ϕ and the *return* ϕ function and use it to enhance PFFS and FFFS to work across methods. Section 3 describes these ϕ functions.
– We then incorporate interprocedural field flow sensitivity into the Whaley-Lam context sensitive analysis and derive an escape analysis based on this. This makes PFFS and FFFS both field flow and context sensitive. Both these analyses are described in Section 4.
– We experimentally study the effects of field flow sensitivity on the timing and precision of a context sensitive pointer and escape analysis. We do this by comparing PFFS and FFFS with two versions of the Whaley-Lam analysis [1]: one of which is flow sensitive for locals (FS) and the other flow insensitive for locals (FIS). Section 5 describes the implementation and gives the results.

2 Intraprocedural Field Flow Sensitivity

2.1 Heap Array SSA and Field Flow Sensitivity

A flow sensitive pointer analysis is costly both in terms of time and space since it has to compute and maintain a separate points-to graph at every program point. One way to reduce the cost is to use the SSA Form [3] for pointer analysis. But translating a program into SSA itself may require pointer analysis, due to the presence of pointer variables in the original program. Hasti and Horwitz [4] give an algorithm for performing flow sensitive analysis using SSA for C, while safely factoring the effect of pointers on the SSA translation. In case of Java, the use of fields gives rise to the same issues as the use of pointer variables in C. However, the normal SSA translation as applied to scalar variables does not give precise

```
public A foo()
{
    S1: u = new A(); // O1
    S2: v = new A(); // O2
    S3: u.f = v;
        ...
    S4: y = helper(u);
    S5: return y;
}

public A helper(A x)
{
    S6: r = new A(); // O3
    S7: ret = x.f;
    S8: ret.f = r;
    S9: return ret;
}
```

```
public void bar()
{
    S10: n = new A(); // O4
    S11: m = new A(); // O5
    S12: o = new A(); // O6
    S13: n.f = m;
         ...
    S14: p = helper(n);
    S15: q = n;
    S16: n.f = o;
         ...
    S17: n.f = m;
    S18: if (...) {
    S19:   q.f = o;
    S20:   q = m;
         }
}
```

Fig. 1. Example to illustrate Field Flow Sensitivity

results for Java fields. Consider the program in Figure 1 and the scalar SSA form of the *bar()* method, shown in Figure 2. It can be inferred from the points-to graph (Figure 3) that q at S15 points to O4 while at S20 it points to O5 (a scalar flow sensitive result), based on the SSA subscripts. However, we cannot infer from this graph that $n.f$ points to O5 at S13 while at S16 it points to O6 (a field flow insensitive result). This is because no distinction is made between the different instances of an object field (f in this case) at different program points. A *field flow sensitive analysis* is one which makes a distinction between field instances at different program points and is more precise than a field flow insensitive analysis.

A field flow sensitive analysis can be obtained by using an extended form of SSA, to handle object fields, called Heap Array SSA. Heap Array SSA is an application of the Array SSA Form [5], initially developed for arrays in C, to Java fields. In the Array SSA form, a new instance of an array is created every time one of its elements is defined. Since the new instance of the array has the correct value of only the element that was just defined, a function called the *define-ϕ* (dϕ) is inserted immediately after the assignment to the array element. The dϕ collects the newly defined values with the values of unmodified elements, available immediately before the assignment, into a new array instance.

The Array SSA form also carries over the *control-ϕ* (cϕ) function from the scalar SSA, inserted exactly at the same location (iterated dominance frontier) as done for scalar SSA. The cϕ merges the values of different instances of an array computed along distinct control paths. Heap Array SSA [2] applies the Array SSA form to Java objects. Accesses to an object field f are modeled by defining a one-dimensional heap array H^f . This heap array represents all instances of the field f that exists on the heap. Heap arrays are indexed by object references. A load of $p.f$ is modeled as read of element $H^f[p]$ and the store of $q.f$ is modeled

```
public void bar()
{
        n0 = new A(); // O4
        m0 = new A(); // O5
        o0 = new A(); // O6
        n1 = n0;
 S13:   n1.f = m0;
          ...
        p0 = helper(n1);
 S15:   q0 = n1;
        n2 = n1;
 S16:   n2.f = o0;
          ...
        n3 = n2;
        n3.f = m0;
        if (...) {
             q1 = q0;
 S19:        q1.f = o0;
 S20:        q2 = m0;
        }
        q3 = mφ(q0, q2);
}
```

Fig. 2. Scalar SSA form of the *bar()* method

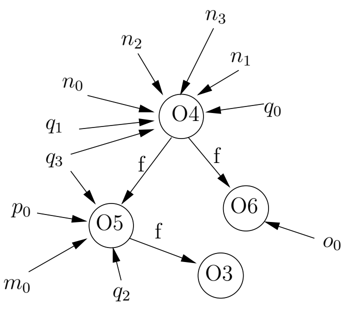


Fig. 3. Points to Graph for Scalar SSA Form of bar()

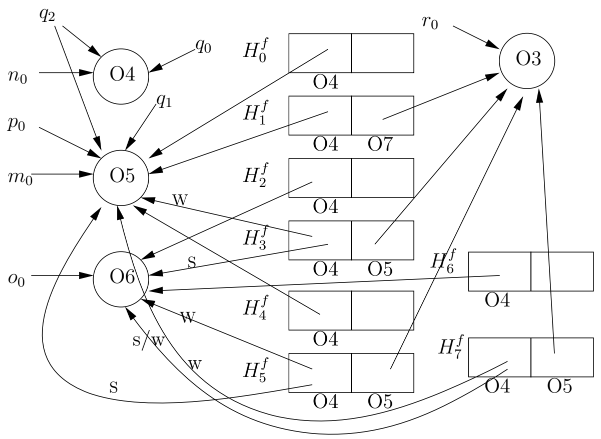


Fig. 4. Points to Relations using Heap Array SSA for *bar()*

as a write of element $H^f[q]$. Figure 5 shows the Heap Array SSA form for the program seen earlier. The mϕ function is the traditional ϕ function used for scalars [3]. Converting a Java program into Heap Array SSA form and running a flow insensitive analysis algorithm on it generates a flow sensitive pointer analysis result for both fields and locals.

2.2 Field Flow Sensitive Analysis as Logic Programs

Pointer analysis can naturally be expressed in a logic programming language [6] like Datalog. The Java statements that affect the points-to relations are given as input relations to the logic program while the points-to relation is the output generated by the analysis. There is one input relation to represent every type of statement in a Java program. Every source statement in the Java program is encoded as a unique tuple (row) in the corresponding input relation. The transfer functions are represented as deduction rules. Whaley and Lam have developed *bddbddb*, a scalable system for solving Datalog programs and implemented a context sensitive version of Anderson's pointer analysis [7] on it. We apply Anderson's analysis

```
public A foo()
{
    u_0 = new A(); // O1
    v_0 = new A(); // O2
    H^f_8[u_0] = v_0;
        ...
    y_0 = helper(u_0);
    H^f_9 = rφ(H^f_12{e_2}, H^f_8);
    return y_0;
}
public A helper(A x)
{
    H^f_10 = iφ(H^f_0{e_1}, H^h_8{e_2});
    r_0 = new A(); // O3
    ret_0 = H^f_10[x];
    H^f_11[ret_0] = r_0;
    H^f_12 = dφ(H^f_11, H^f_10);
    return ret_0;
}
```

```
public void bar()
{
       n_0 = new A(); // O4
       m_0 = new A(); // O5
       o_0 = new A(); // O6
 S13:  H^f_0[n_0] = m_0;
         ...
       p_0 = helper(n_0);
       H^f_1 = rφ(H^f_12{e_1}, H^f_0);
       q_0 = n_0;
 S16:  H^f_2[n_0] = o_0;
       H^f_3 = dφ(H^f_2, H^f_1);
         ...
       H^f_4[n_0] = m_0;
       H^f_5 = dφ(H^f_4, H^f_3);
       if (...) {
 S19:      H^f_6[q_0] = o_0;
           H^f_7 = dφ(H^f_6, H^f_5);
           q_1 = m_0;
       }
       q_2 = mφ(q_1, q_0);
       H^f_8 = cφ(H^f_7, H^f_5);
}
```

Fig. 5. Heap Array SSA Form for the program in Figure 1

over the Heap Array SSA form of a Java program to obtain a field flow sensitive analysis. The precision of the resulting analysis is as good as any field flow sensitive analysis that is based on points-to graphs [8][9]. The main advantage of using Heap Array SSA is that it obviates the need to maintain a separate points-to graph at every program point and thereby effecting a more scalable analysis.

We formulate two variants of this analysis in Datalog: partial field flow sensitive analysis (PFFS) and full field flow sensitive analysis (FFFS). All heap objects are abstracted by their allocation sites. Two points-to sets, *vPtsTo* and *hPtsTo*, are associated with scalar variables and heap array elements (heap arrays themselves are indexed by objects) respectively. These sets hold the objects pointed to by them. At the end of analysis, *vPtsTo* and *hPtsTo* together give a field flow sensitive points-to result. We use Whaley and Lam's notation for logic programs [1]. V and H represent the domain of scalars and heap objects (object numbers) respectively. F is the domain of all fields. The numeric subscripts inserted by the Heap Array SSA transformation are represented from N, the set of natural numbers. The relations used in our analysis have an attribute to accommodate the SSA numbers of the heap arrays. For local variables, the SSA subscripts are a part of the variable name itself. Table 1 lists for every input relation, the tuple representation of a particular source statement. For an output relation, it specifies the tuple representation for a particular element of the derived points-to set.The first four deduction rules of the analysis capture the effect of *new*, *assign*, *load* and *store* statements and are very similar to the rules for the non-SSA form, the only difference being the SSA numbers for heap arrays:

$$vPtsTo(v_1, h) \quad :- \quad new(v_1, h) \tag{1}$$
$$vPtsTo(v_1, h) \quad :- \quad assign(v_1, v_2),\ vPtsTo(v_2, h) \tag{2}$$

$$vPtsTo(v_1, h_2) \quad :- \quad load(v_1, f, s_0, v_b),\ vPtsTo(v_b, h_1),\ hPtsTo(f, s_0, h_1, h_2) \tag{3}$$
$$hPtsTo(f, s_0, h_1, h_2) \quad :- \quad store(v_1, f, s_0, v_b),\ vPtsTo(v_b, h_1),\ vPtsTo(v_1, h_2) \tag{4}$$

The semantics of the rules are as follows :

- **Rule (1) for new** $v_1 = new\ \ h()$: Creates the initial points-to relation for the scalar variables.
- **Rule (2) for assign** $v_1 = v_2$: Updates the points-to set for v_1 based on the inclusion property: $points - to(v_2) \subseteq points - to(v_1)$.
- **Rule (3) for load** $v_1 = H^f_{s_0}[v_b]$: Updates the points-to set for v_1 using the points-to set of $H^f_{s_0}[h_1]$ for every object h_1 that is pointed to by the index variable v_b of the Heap array instance $H^f_{s_f}$.
- **Rule (4) for store** $H^f_{s_0}[v_b] = v_1$: Acts similar to rule (3), the data flow being in the opposite direction in this case.

The next two rules correspond to the $c\phi$ and the $m\phi$ statement:

$$hPtsTo(f, s_0, h_1, h_2) \quad :- \quad cphi(f, s_0, s_1),\ hPtsTo(f, s_1, h_1, h_2) \tag{5}$$
$$vPtsTo(v_0, h) \quad :- \quad mphi(v_0, v_1),\ vPtsTo(v_1, h) \tag{6}$$

The semantics of these rules are:

- **Rule (5) for cϕ:** $H^f_{s_0} = c\phi(H^f_{s_1}, H^f_{s_2}, ..., H^f_{s_n})$: The c$\phi$ statement is represented as a set of tuples (f, s_0, s_i) $\forall i$ such that $1 \leq i \leq n$ in the *cphi* relation since all the arguments of cϕ are symmetric with respect to the lhs heap array instance. The effect of the rule (5) is to merge the points-to sets corresponding to all valid object indices of its arguments into the rhs heap array instance.
- **Rule (6) for mϕ:** [1] $v_0 = m\phi(v_1, ..., v_n)$ Performs the merge of points-to sets for scalars.

To complete the analysis, we need to add rules corresponding to the dϕ statement. Based on the type of rules for modeling dϕ, we distinguish two types of field flow sensitivity:

Partial Field Flow Sensitivity (PFFS). We define a partial field flow sensitive analysis as one that performs only weak updates to heap array elements. To obtain PFFS, the rules required to model dϕ statement are simple and are

[1] In the implementation, this rule is replaced by rule (2) for assigns, since a mϕ can be modeled as a set of assignment statements.

Table 1. Relations used in the Field Flow Sensitive Analysis

Source Statement/ Pointer Semantics	Tuple(s) Representation	Relations	Type
$v_1 = new\ \ h()$	(v_1, h)	$new(v$: V, h: H)	input
$v_1 = v_2$	(v_1, v_2)	$assign(v_1$: V, v_2: V)	input
$v_2 = H^f_{s_0}[v_1]$	(v_2, f, s_0, v_1)	$load(v$: V, f: F, s: N, b: V)	input
$H^f_{s_0}[v_1] = v_2$	(v_2, f, s_0, v_1)	$store(v$: V, f: F, s: N, b: V)	input
$H^f_{s_0} = d\phi(H^f_{s_1}, H^f_{s_2})$	(f, s_0, s_1, s_2)	$dphi(f$: F, s_0: N, s_1: N, s_2: N)	input
$H^f_{s_0} = c\phi(H^f_{s_1}, ..., H^f_{s_n})$	(f, s_0, s_i) $\forall i$ such that $1 \leq i \leq n$	$cphi(f$: F, s_0: N, s_1: N)	input
$v_0 = m\phi(v_1, ..., v_n)$	(v_0, v_i) $\forall i$ such that $1 \leq i \leq n$	$mphi(v_0$: V, v_1: V)	input
$v_0 \rightarrow h$	(v_0, h)	$vPtsTo(v$: V, h: H)	output
$H^f_{s_0}[h_1] \rightarrow h_2$	(f, s_0, h_1, h_2)	$hPtsTo(f$: F, s_0: N, h_1: H, h_2: H)	output

similar to that for the cϕ statement. Without strong updates, useful information can still be obtained since the points-to relation of heap arrays that appear at a later point in the control flow do not interfere with the points-to relations of the heap array at the current program point. The rules for achieving PFFS are:

$$hPtsTo(f, s_0, h_1, h_2) \quad :- \quad dphi(f, s_0, s_1, -),\ hPtsTo(f, s_1, h_1, h_2) \tag{7}$$
$$hPtsTo(f, s_0, h_1, h_2) \quad :- \quad dphi(f, s_0, -, s_2),\ hPtsTo(f, s_2, h_1, h_2) \tag{8}$$

The semantics of these rules are:

– **Rules (7) and (8) for dϕ:** $H^f_{s_0} = d\phi(H^f_{s_1}, H^f_{s_2})$: Merge the points-to sets corresponding to *all* the heap array indices of both the argument heap arrays $H^f_{s_1}$ and $H^f_{s_2}$ and gather them into the lhs heap array instance $H^f_{s_0}$. As no pointed object is ever evicted (killed) from a heap array at a store, only a weak update to fields is done.

Full Field Flow Sensitivity (FFFS). We define a fully field flow sensitive analysis as one that performs strong updates to heap array elements. Hence, FFFS is more precise than PFFS. However, a strong update can be applied at a store statement $v_b.f = v_2$ only under two conditions:

1. v_b points to a single abstract heap object that represents only one concrete object at runtime.
2. The method in which the abstract object is allocated should not be a part any loop or recursive call chain.[2]

[2] This is because we do not have any information about the predicate conditions for loops/recursive method invocations and have to conservatively infer that an object can be allocated more than once, preventing the application of a strong update.

One way to model the dϕ statement to allow for strong updates is by the following rules:

$$\begin{aligned}
hPtsTo(f, s_0, h_1, h_2) \quad &:- \quad dphi(f, s_0, s_1, -),\ hPtsTo(f, s_1, h_1, h_2) &(9)\\
hPtsTo(f, s_0, h_1, h_2) \quad &:- \quad store(-, f, s_1, v_b),\ nonsingular(v_b),\\
&\quad\quad dphi(f, s_0, s_1, s_2),\ hPtsTo(f, s_2, h_1, h_2),\\
&\quad\quad hPtsTo(f, s_1, h_1, -) &(10)\\
hPtsTo(f, s_0, h_1, h_2) \quad &:- \quad mayBeInLoop(h_1),\ dphi(f, s_0, s_1, s_2),\\
&\quad\quad hPtsTo(f, s_2, h_1, h_2),\ hPtsTo(f, s_1, h_1, -) &(11)\\
hPtsTo(f, s_0, h_1, h_2) \quad &:- \quad dphi(f, s_0, s_1, s_2),\ hPtsTo(f, s_2, h_1, h_2),\\
&\quad\quad !commonIndex(f, s_1, s_2, h_1) &(12)\\
nonsingular(v_1) \quad &:- \quad vPtsTo(v_1, h_1),\ vPtsTo(v_1,\ h_2), h_1 != h_2 &(13)
\end{aligned}$$

The semantics of these rules are:
Rules (9), (10), (11), (12) and (13) for dϕ: $H^f_{s_0} = d\phi(H^f_{s_1}, H^f_{s_2})$: Whenever a dϕ statement is encountered, the points-to sets for lhs heap array instance $H^f_{s_0}$ is constructed from its arguments $H^f_{s_1}$ and $H^f_{s_2}$ as follows:

– All the points-to sets for object indices of $H^f_{s_1}$ are carried over to $H^f_{s_0}$. The ordering of the arguments for the dϕ is important here: The heap array instance $H^f_{s_1}$ corresponds to the one that is defined in the store statement immediately before this dϕ statement. Rule (9), which performs the derivation of $H^f_{s_0}$ from $H^f_{s_1}$, depends on this ordering to work.
– The points-to sets from $H^f_{s_2}$, for object indices common to $H^f_{s_1}$ and $H^f_{s_2}$, are conditionally carried over to $H^f_{s_0}$ using rules (10) and (11). These rules represent the negation of the conditions required to satisfy a strong update (kill) for a store statement. The *nonsingular* relation determines whether a variable may point to more than one heap object and is derived using rule (13). When the base variable v_b of the store statement is in the *nonsingular* relation, the points-to sets from $H^f_{s_2}$ go into $H^f_{s_0}$, using rule (10). The *mayBeInLoop* relation, on the other hand, is an input relation, which represents all the allocation sites which may be executed more than once. The heap objects abstracted at these sites may not represent a single runtime object. We compute this input relation using a control flow analysis provided by *SOOT*. Whenever a target heap object h_1 of a store is in the *mayBeInLoop* relation, the points-to sets from $H^f_{s_2}$ go into $H^f_{s_0}$, using rule (11).
– The points-to sets from $H^f_{s_2}$, for object indices not common to $H^f_{s_1}$ and $H^f_{s_2}$, are carried over to $H^f_{s_0}$ using rule (12). The computation of the common indices itself requires a pointer analysis due its inherent recursive nature.
– **Strong Updates and Non stratified logic programs:** Consider, for a moment, the following rule as a replacement for (12):

$$\begin{aligned}
hPtsTo(f, s_0, h_1, h_2) \quad &:- \quad dphi(f, s_0, s_1, s_2),\ hPtsTo(f, s_2, h_1, h_2),\\
&\quad\quad !hPtsTo(f, s_1, h_1, -)
\end{aligned}$$

This rule uses recursion with negation on *hPtsTo* relation to compute itself. This rule would result in a *non-stratified* Datalog program, which is currently not supported by *bddbddb*. The way *bddbddb* evaluates Datalog programs is by constructing a *predicate dependency graph* (PDG), where each node represents a relation and an edge $a \rightarrow b$ exists if a is the head relation of a rule which has b as a subgoal relation. Also, an edge is labeled as negative if the subgoal relation has a negation in the rule. The PDG is then divided into different strongly connected components (SCC) and the relations within each SCC are wholly computed by doing a fixed point iteration over the rules representing the edges within the SCC. The whole program is then evaluated in the topological ordering of these SCCs. A Datalog program becomes non-stratified if there exists a SCC with a negative edge. In this case, there is a cycle from *hPtsTo* to itself. XSB [10], a system which supports non-stratified programs using well-founded semantics,[3] operates at the tuple level and is not as scalable as *bddbddb* which operates on complete relations by representing them as BDDs and using efficient BDD operations. Our approach here is to pre-compute the *commonIndex* relation, which represents an over-approximation of the set of common object indices for the two argument heap arrays $H^f_{s_2}$ and $H^f_{s_1}$ of the dϕ and get a stratified Datalog program. The *commonIndex* relation is pre-computed using a field flow insensitive analysis pass, PFFS, in our case:

$$\begin{aligned} commonIndex(f, s_1, s_2, h_1) \quad &:- \quad dphi(f, -, s_1, s_2),\ hPtsTo(f, s_1, h_1, -), \\ &\qquad hPtsTo(f, s_2, h_1, -) \end{aligned} \tag{14}$$

Consider the points-to relations obtained by the field flow sensitive analysis for the *bar()* method as seen in Figure 4. The edges labeled w (weak) are the additional edges inferred by PFFS which FFFS does not infer. Both FFFS and PFFS infer the s (strong) edges. Both the analyses infer that at S13, $n.f$ points to O5 ($n_0 \rightarrow O4$ and $H^f_0[O4] \rightarrow O5$) while at S16, $n.f$ points to O6 ($n_0 \rightarrow O4$ and $H^f_2[O4] \rightarrow O6$), which is a field flow sensitive result. However, if there was a reference to $n.f$ after S16, PFFS would infer that $n.f$ may point to either O5 or O6 (due to the w edge: $H^f_3[O4] \rightarrow O6$) while FFFS would say that $n.f$ may point to only O6 (the single s edge: $H^f_3[O4] \rightarrow O6$).

3 Interprocedural Field Flow Sensitivity: iϕ and rϕ

To obtain field flow sensitivity in the presence of method calls, we have to take into account: (a) Effects of field updates made by a called method visible to the caller at the point of return (b) Flow of correct object field values into a called method depending on the invocation site from where it is called. We introduce two new ϕ functions to extend the Heap Array SSA Form interprocedurally: (a) The invocation ϕ function, iϕ (b) The return ϕ function, rϕ.

[3] It provides a form of 3-valued evaluation for logic programs.

The iϕ function models the flow of values into a method corresponding to the invocation site from where it is called. It selects the exact points-to set that exists at the invocation site (at the point of call), on the basis of the invocation edge in the call graph. This is in contrast with the $c\phi$ function which merges the points-to sets flowing via different control flow paths. The iϕ function has the following form:

$$H^f_{s_{in}} = i\phi(H^f_{s_1}\{e_1\}, H^f_{s_2}\{e_2\}, ..., H^f_{s_n}\{e_n\})$$

where $H^f_{s_1}, H^f_{s_2}, ..., H^f_{s_n}$ are the heap array instances that *dominate*[4] the point of call and $e_1, e_2, ..., e_n$ are the corresponding invocation edges in the call graph.

The rϕ function models the merge of the points-to sets of the heap array that existed before the call and the points-to set of the heap arrays that were modified by the call. The updated heap array after the call models the effect of all the methods transitively called in the call chain. It has the following form:

$$H^f_{s_r} = r\phi(H^f_{s_1}\{e_1\}, H^f_{s_2}\{e_2\}, ..., H^f_{s_n}\{e_n\}, H^f_{s_{local}})$$

The presence of more than one edge in the rϕ function is due to virtual method invocations. $H^f_{s_i}$, $\forall i$ such that $1 \leq i \leq n$ is the dominating heap array instance that is in effect at the end of the called method, with corresponding invocation edge e_i. Thus, for every possible concrete method that can be called at the call site, rϕ collects the heap array instances for the field f that dominate the end of the called method and merges the points-to sets into $H^f_{s_r}$. Those objects whose field f has not been modified get their points-to sets into $H^f_{s_r}$ from $H^f_{s_{local}}$, the heap array instance for f that dominates the immediate program point before the call site in the calling method.

The $i\phi$'s are placed at the entry point of a method. Only those heap arrays that are *used* or *modified* in the current method and all the methods it calls transitively require an $i\phi$ at the entry point. Similarly, an $r\phi$ is required only for those heap arrays which are *modified* transitively by a method call. The list of heap arrays for which $r\phi$ and $i\phi$ are required can be determined while performing a single traversal (in reverse topological order of nodes of the call graph) on the interprocedural control flow graph of the program. For the example in Figure 5, an $i\phi$ is placed for f in the *helper()* while rϕ's are placed after calls to *helper()* in *foo()* and *bar()*. The lhs heap arrays of these ϕ functions are also renamed as a part of Cytron's SSA renaming step [3].

Although the previous step would have determined the placement of the $i\phi$ and $r\phi$, still we have to determine their arguments and rename the heap array instances used in the arguments. This is achieved by plugging in the heap array instances that dominate the *point of call* and those that dominate the *callee's exit points* into the arguments of $i\phi$ and $r\phi$ respectively. For the example in Figure 5, the arguments of the iϕ in *helper*() method are H^f_0 and H^f_8, the heap arrays that dominate

[4] By definition of the *dominates* relation [3], there is only one dominating heap array instance at any program point for every field in the Heap Array SSA form.

Table 2. Additional Relations for Interprocedural Field Flow and Context Sensitivity

Source Statement/ Pointer Semantics	Tuple(s) Representation	Relation	Type
An invocation i from context c_1 to m in context c_2	(c_1, i, c_2, m)	$IE_c(c_1$: C, i: I, c_2: C, m: M)	input
$H^f_{s_{in}} = i\phi(H^f_{s_1}\{e_1\}, ..., H^f_{s_n}\{e_n\})$	(f, s_{in}, s_i, e_i) $\forall i$ such that $1 \leq i \leq n$	$iphi(f$: F, s_{in}: Z, s_i: Z, e_i: I)	input
$H^f_{s_r} = r\phi(H^f_{s_1}\{e_1\}, ..., H^f_{s_n}\{e_n\}, H^f_{s_l})$	(f, s_r, s_l, s_i, e_i) $\forall i$ such that $1 \leq i \leq n$	$rphi(f$: F, s_r: Z, s_l: Z, s_i: Z, e_i : I)	input
In context c_1, $v_1 \to h$	(c_1, v_1, h)	$vPtsTo(c$: C, v: V, h: H)	output
In context c_1, $H^f_{s_0}[h_1] \to h_2$	(c_1, f, s_0, h_1, h_2)	$hPtsTo(c$: C, f: F, s_f: N, h_1: H, h_2: H)	output

the points, in *bar()* and *foo()* respectively, at which *helper*() is called. Similarly the argument of the rϕ's for f is H^f_{12}, the dominating heap array instance in *helper*() at the point of return. The overall placement of the *phi* functions ($m\phi$, $d\phi$, $c\phi$, $i\phi$ and $r\phi$) and their renaming are performed in the following order:

1. Place $d\phi$, $c\phi$ and $m\phi$ functions using dominance frontiers as in [3]
2. Place the $r\phi$ and $i\phi$ functions.
3. Apply Cytron's Algorithm [3] to rename the Heap Array Instances (results of $d\phi$, $c\phi$, $r\phi$, $i\phi$ and arguments of $d\phi$ and $c\phi$) and local variables (results and arguments of $m\phi$).
4. Use the dominating heap array instances at exit points of methods and call sites to plug in the arguments for $r\phi$ and $i\phi$.

4 Combined Field Flow and Context Sensitivity

4.1 Pointer Analysis

Using the iϕ and rϕ functions, we incorporate interprocedural field flow sensitivity into the Whaley-Lam context sensitive analysis [1] algorithm. We describe only those relations (Table 2) and rules that pertain to the iϕ and rϕ statements. The rest of the relations and rules are context sensitive extensions of those mentioned in Section 2.2 and those for parameter and return value bindings, invocation edge representations [1]. I is the domain of invocation edges, M represents all the methods and C is the domain of context numbers. Two main relations new to this analysis are *iphi* and *rphi*, while *vPtsTo* and *hPtsTo* now have an additional attribute for context numbers. The IE_c relation represents context sensitive invocation edges, computed using *SOOT*'s pre-computed call graph and context numbering scheme of Whaley and Lam [1]. In this scheme, every method is assigned a unique context number for every distinct calling context.[5] The deduction rules and their semantics are as follows:

$$hPtsTo(c_2, f, s_i, h_1, h_2) \;:-\; iphi(f, s_i, s_1, i), IE_c(c_1, i, c_2, -), hPtsTo(c_1, f, s_1, h_1, h_2). \tag{15}$$

$$hPtsTo(c_1, f, s_r, h_1, h_2) \;:-\; rphi(f, s_r, -, s_i, i), IE_c(c_1, i, c_2, -), hPtsTo(c_2, f, s_i, h_1, h_2). \tag{16}$$

$$hPtsTo(c_1, f, s_r, h_1, h_2) \;:-\; rphi(f, s_r, s_l, -, i), IE_c(c_1, i, -, -), hPtsTo(c_1, f, s_l, h_1, h_2). \tag{17}$$

– **Rule (15) for** $i\phi$: This rule models the effect of the $i\phi$. When there is a change in context from c_1 to c_2 due to an invocation i, the heap array instance for a field f in context c_2 (the lhs of the $i\phi$ with SSA number s_i) inherits its points-to set from the heap array instance in c_1 before the call was made (argument s_1 corresponding to the invocation edge i in the $i\phi$ statement). The presence of IE_c makes sure that points-to set of multiple calling contexts don't interfere with each other.
– **Rules (16) and (17) for** $r\phi$: These rules are similar to Rules (7) and (8) that model the effect of $d\phi$ statement in PFFS. Rule (16) makes sure that the points-to sets of the heap array instance (s_i in context c_2) from a virtual method invocation (invocation edge i) are merged into that of lhs heap array instance (s_r) in context c_1. Rule (17) ensures that the lhs heap array instance gets the points-to sets from the local heap array instance that dominates the call site (s_l in context c_1).

For the only iϕ in our example, H_{10}^f in *helper()* inherits its points to sets from H_0^f along edge e_1 (called from *bar()*) and from H_8^f along edge e_2 (called from *foo()*) in separate contexts. Hence ret_0 points to O5 and O2 in two distinct contexts and consequently, y_0 and p_0 point to O2 and O5 respectively.

4.2 Escape Analysis for Methods

Escape Analysis [8][9] is a compiler analysis technique which identifies objects that are local to a particular method. For such objects, the compiler can perform stack-allocation, which helps to speed up programs by lessening the burden on the garbage collector. Escape analysis works by determining whether an object may escape a method and if an object does not escape a method ("captured"), it can be allocated on the method's stack frame. Adding a few more relations (Table 3) and rules to the analysis of Section 4.1 gives an escape analysis. We encode the heap array instances that dominate the exit points of methods in

[5] For eg, if a call to method *helper()* by method *bar()* is represented by an invocation edge e_1 in the call graph, and *bar*() is in a calling context with context number c_1 while *helper*() is in a context numbered c_2, this invocation would be represented by the tuple $IE_c(c_1, e_1, c_2, helper)$.

Table 3. Additional Relations for Escape Analysis

Relation	Type	Tuple Semantics
dominatingHA(m: M, f: F, s: Z)	input	Heap Array Instance s for field f dominates exits of method m
$formal(m : M, z : Z, v : V)$	input	Formal parameter z of method m is represented by variable v
threadParam(v: V)	input	v is passed as parameter to a thread
callEdge(m_1: M, m_2: M)	input	A call edge $m_1 \rightarrow m_2$ exists in the call graph
allocated(h: H, m: M)	input	Object h allocated within m
classNode(c: V, h: H)	input	Class c is given a heap number h
escapes(h: H, m: M)	output	Object h escapes m, h need not be allocated within m
aEscapes(h: H, m: M)	output	Object h, allocated within m, escapes m
captured(h: H, m: M)	output	Object h, allocated within m, does not escape m
callerCaptured(h: H, m: M)	output	Object h, allocated within m, escapes m, but does not escape an immediate caller of m

a relation (*dominatingHA*) and every class is uniquely numbered in the heap objects' domain.

$$\begin{aligned} escapes(h_2, m) \quad &:- \quad dominatingHA(m, f, s_0), classNode(-, h_1), \\ &\quad hPtsTo(-, f, s_0, h_1, h_2) && (18) \\ escapes(h, m) \quad &:- \quad vPtsTo(-, v, h), formal(m, -, v). && (19) \end{aligned}$$

$$\begin{aligned} escapes(h, m) \quad &:- \quad return(m, v), vPtsTo(-, v, h). && (20) \\ escapes(h, -) \quad &:- \quad threadParam(v), vPtsTo(-, v, h). && (21) \\ escapes(h_2, m) \quad &:- \quad escapes(h_1, m), hPtsTo(-, f, s_0, h_1, h_2), \\ &\quad dominatingHA(m, f, s_0). && (22) \\ aEscapes(h, m) \quad &:- \quad escapes(h, m), allocated(h, m). && (23) \\ captured(h, m) \quad &:- \quad !aEscapes(h, m), allocated(h, m). && (24) \\ callerCaptured(h, m_1) \quad &:- \quad !escapes(h, m_1), aEscapes(h, m_2), \\ &\quad callEdge(m_1, m_2). && (25) \end{aligned}$$

The semantics of these rules are as follows :

- **Rules (18) to (21) : Direct Escape**: These rules determine the objects that directly escape a method m: objects whose reference is stored in a static class variable (Rule 18), objects representing the parameters (Rule 19), objects returned from a method (Rule 20) and thread objects and objects passed to thread methods (Rule 21)
- **Rules (22) to (25): Indirect Escape, Capture and Caller Capture**: Rules (22)-(24) compute the indirectly escaping objects, ie, those reachable via a sequence of object references from a directly escaped object. The remaining objects are '*captured*', ie, those that are inaccessible outside their method of allocation. Finally, Rule (25) computes the *caller-captured* objects, which escape their method of allocation but are captured within a caller method. Such caller-captured objects can be stack allocated in the caller's stack.

Name	Description	Byte codes
jip	Java Interactive Profiler	210K
umldot	UML Diagram Creator	142K
jython	Python Interpreter	295K
jsch	Implementation of SSH	282K
java_cup	Parser Generator	152K
jlex	Lexical Analyzer Generator	91K
check	Checker for JVM	46K
jess	Java Expert Shell	13K
cst	Hashing Implementation	32K
si	Small Interpreter	24K
compress	Modified Lampel-Ziv method	21K
raytrace	Ray tracer	65K
db	Memory Resident Database	13K
anagram	Anagram Generator	9K
mtrt	A variant of raytrace	1K

Fig. 6. Benchmark Programs used

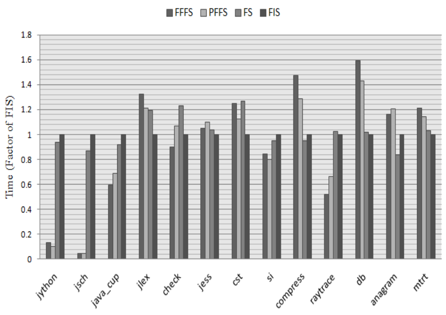


Fig. 7. Time of Analysis

5 Experimental Results

We have implemented both PFFS and FFFS on the *bddbddb* system, using *SOOT* as the front end to generate Heap Array SSA and the input relations. We ran our analyses on some of the popular Java programs from SourceForge and SPEC JVM 98 benchmark suite (Figure 6). All the programs were run in whole program mode in with a precomputed call graph in *SOOT*.[6] The analyses were done on 4 CPU 3.20 GHz Intel Pentium IV PC with 2 GB of RAM running Ubuntu Linux.

We compare our analyses with two field flow insensitive versions of the Whaley-Lam analysis [1]: one of which is flow sensitive for locals (FS) and the other flow insensitive for locals (FIS). The Whaley-Lam analysis is context sensitive. The FS analysis uses the scalar SSA form to obtain flow sensitivity for locals while FIS does not make use of SSA. The comparisons are based on: (a) Time of analysis (b) Precision in terms of size of points-to sets at load/store statements (c) Number of objects found to be captured in the escape analysis. Figure 7 shows the relative time taken for the four analyses, normalized with respect to FIS. Figure 8 shows the average number of objects pointed to by an object field $x.f$ at a load $y = x.f$ (which we call the *loadPts* set), again as a factor of FIS. This is computed based on the *vPtsTo* relation for x at the load and the *hPtsTo* relation for objects pointed to by x. The average number of objects pointed to by a scalar variable x at a store $x.f = y$ (the *storePts* set), inferred by each analysis is shown in Figure 9.

The time taken by the field flow sensitive versions are comparable to FS and FIS for most programs, while doing much better in some cases. Also, for two

[6] Although we use a precomputed call graph here, this analysis can be combined with a on-the-fly call graph construction using the techniques employed in [1].

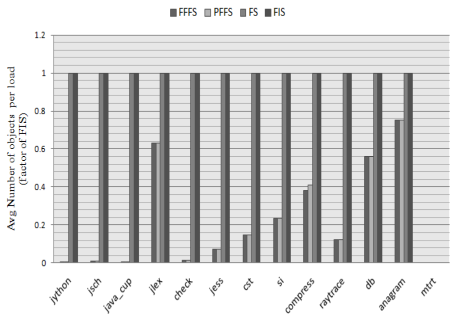


Fig. 8. Average Size of *loadPts* set

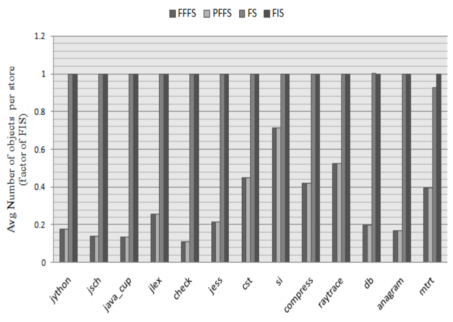


Fig. 9. Average Size of *storePts* set

programs, *jip* and *umldot*, the JVM ran out of memory while running FIS and FS. In terms of precision, both PFFS and FFFS reduce the size of *loadPts* set to about 23% of the size computed by FIS, averaged over all programs. The size of the *storePts* set is reduced to about 30% of the size computed by FIS.

Three observations can be made from these plots: Firstly, these results illustrate the importance of field flow sensitivity in a context sensitive analysis, especially in programs written in a object oriented programming language like Java where short and frequently invoked methods are the common case [11]. A field flow sensitive analysis takes advantage of longer interprocedural program paths that have been identified as distinct from each other by a context sensitive analysis. In the absence of field flow sensitivity, even though a context sensitive analysis identifies longer distinct interprocedural paths, the points-to sets of object fields at various points along the path are merged. Secondly, programs for which the size of *loadPts* as computed by PFFS is less than 10% as that computed by FIS, the analysis time for PFFS is also much less than FIS. Hence field flow sensitivity not only helps in getting a precise pointer analysis result, but also helps in reducing analysis times to some extent, by avoiding the computation of spurious points-to relations (*hPtsTo*) across methods. Finally, as is evident from the size of *loadPts* and *storePts* computed by FFFS and PFFS, there is very little gain in precision by using FFFS as compared to PFFS. This counter-intuitive result can be explained by the following observation: In the absence of (a) support for non stratified queries and (b) an accurate model of the heap, the conditions for strong updates (all of which are required for correctness) are very strong and their scope is limited to only a few field assignments that always update a single runtime object and that too at most once. Although the developers of *bddbddb* did not find the need for non stratified queries for program analysis,[7] the use of Heap array SSA to perform aggressive strong updates does illustrate an instance where non stratified queries are of

[7] We quote, from [12]: "In our experience designing Datalog programs for program analysis, we have yet to find a need for non-stratifiable queries".

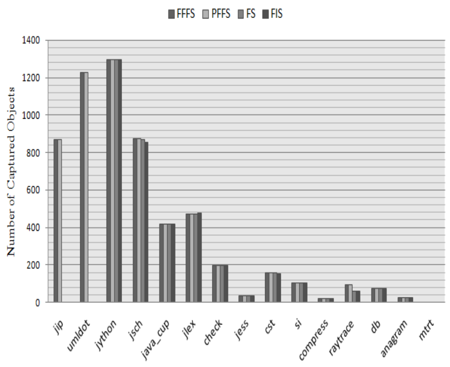


Fig. 10. No. of captured objects

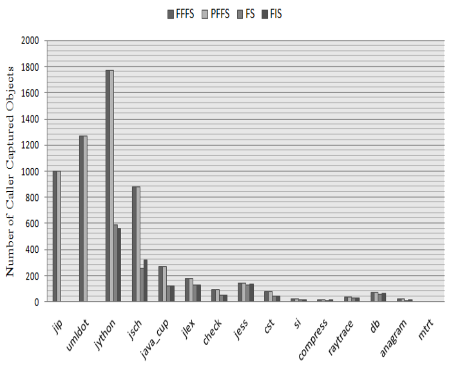


Fig. 11. No. of recaptured objects

importance to a program analysis. In addition, an accurate model of the heap can greatly help in improving precision. A shape analysis builds better abstractions of structure of the heap and enables strong updates much more freely than is currently possible. One of the most popular shape analysis algorithms [13] is based on three-valued logic and to formulate this analysis in the logic programming paradigm, we might need to adopt a different kind of semantics like the well founded semantics, as done in XSB [10], which also handles non-stratified queries.

The *captured* relation sizes are shown in Figure 10. This relation computes the captured state of an object with respect to its method of allocation. As seen from the graph, the number of objects captured in their method of allocation is almost the same for the field flow sensitive and insensitive versions. This is surprising given that PFFS/FFFS have an extra level of precision and hence should have discovered more captured objects. Figure 11 compares the number of *caller-captured* objects (ie, objects that escape their method of allocation, but are caught in their immediate caller) discovered by the four analyses. PFFS improved the number of caller-captured objects by an average of about 71% compared to FIS (again there was not much gain in using FFFS over PFFS). Such caller-captured objects can be stack allocated in the calling method's stack. This is beneficial when coupled with partial specialization of Java methods [14]. Since the computation of *caller-captured* objects takes into account the flow sensitivity of field assignments across two methods, PFFS gives a better *caller-captured* set than *captured*, especially in the presence of context sensitivity. This is because an object that escapes its method of allocation can be captured in more than one of its callers (each in a separate calling context) and an interprocedural field flow sensitive analysis with its dominating heap array information can help in discovering such objects better than a field flow insensitive analysis with no control flow and dominance information.

6 Related Work

Our work is inspired by Whaley and Lam's work on context sensitive analysis [1] and work by Fink et al. [2] on Heap Array SSA. Whaley and Lam also specify a thread escape analysis while we have used the pointer analysis results to infer a *method* escape analysis. One of the earliest context-insensitive and flow-insensitive pointer analysis was due to Anderson [7] which is inclusion based, solved using subset constraints. The analysis of Emami et al [15], formulated for C, is both context and flow sensitive. It computes both *may* and *must* pointer relations and context sensitivity is handled by regarding every path in the call graph as a separate context (cloning). Our analysis is a *may* pointer analysis while for context sensitivity the cloned paths are represented by BDDs using *bddbddb*. Reps describes techniques for performing interprocedural analysis using logic databases and gives the relation between context-free reachability, logic programs and constraint based analyses [6]. All program analysis problems whose logic programs are chain programs are equivalent to a context-free reachability problem. Sridharan and Bodik express a context sensitive, flow insensitive pointer analysis for Java as a context-free reachability (CFL) problem [16]. We could not use CFL due to the presence of logic program rules which were not chain rules, for eg, that for *dphi* function rule that adds the field flow sensitivity.

Milanova et al [17] propose *object sensitivity* as a substitute for context sensitivity using call strings. Object names are used, instead of context numbers, to distinguish the pointer analysis results of a method which can be invoked on them. Object names represent an abstraction of a sequence of object allocation sites on which methods can be invoked. Their analysis is flow insensitive. The field flow sensitive portion of our analysis could be used with object sensitivity instead of context sensitivity. Whaley and Rinard's escape analysis [8], similar to Choi et al's analysis [9], maintains a points-to escape graph at every point in the program and hence is fully field flow sensitive. However, complete context and field flow sensitivity is maintained only for objects that do not escape a method. The pointer information for objects that escape a method are merged.

7 Conclusions

In this paper, we presented two variants of a field flow sensitive analysis for Java using the Heap Array SSA Form in Datalog. We have extended the Heap Array SSA form interprocedurally to obtain field flow sensitivity in the presence of context sensitivity and derived an escape analysis based on this. We have implemented our analysis using *SOOT* and *bddbddb*. Our results indicate that partial field flow sensitivity obtains significant improvements in the precision of a context sensitive analysis and helps to identify more captured objects at higher levels in the call chain. Strong updates do not seem to lead to any further gain in precision in current system. The running times of our analysis are comparable to a field flow insensitive analysis, while in some cases running much faster than the latter.

Acknowledgments

We thank John Whaley for the *bddbddb* system and clarifying some of our doubts regarding his paper. We thank the people at McGill for the *SOOT* framework.

References

1. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Programming language design and implementation, pp. 131–144 (2004)
2. Fink, S.J., Knobe, K., Sarkar, V.: Unified analysis of array and object references in strongly typed languages. In: Static Analysis Symposium, pp. 155–174 (2000)
3. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13(4), 451–490 (1991)
4. Hasti, R., Horwitz, S.: Using static single assignment form to improve flow-insensitive pointer analysis. In: Programming language design and implementation, pp. 97–105 (1998)
5. Knobe, K., Sarkar, V.: Array SSA form and its use in parallelization. In: Symposium on Principles of Programming Languages, pp. 107–120 (1998)
6. Reps, T.W.: Program analysis via graph reachability. In: International Logic Programming Symposium, pp. 5–19 (1997)
7. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (May 1994)
8. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: Object-oriented programming, systems, languages, and applications, pp. 187–206 (1999)
9. Choi, J.D., Gupta, M., Serrano, M., Sreedhar, V.C., Midkiff, S.: Escape analysis for Java. In: Object-oriented programming, systems, languages, and applications, pp. 1–19 (1999)
10. Sagonas, K., Swift, T., Warren, D.S.: XSB as an efficient deductive database engine. In: International conference on Management of data, pp. 442–453 (1994)
11. Budimlic, Z., Kennedy, K.: Optimizing Java: theory and practice. Concurrency: Practice and Experience 9(6), 445–463 (1997)
12. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using Datalog and binary decision diagrams for program analysis. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, Springer, Heidelberg (2005)
13. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3–valued logic. In: Symposium on Principles of Programming Languages, pp. 105–118 (1999)
14. Schultz, U.P., Lawall, J.L., Consel, C.: Automatic program specialization for Java. ACM Trans. Program. Lang. Syst. 25(4), 452–499 (2003)
15. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Programming language design and implementation, pp. 242–256 (1994)
16. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for Java. In: Programming language design and implementation, pp. 387–400 (2006)
17. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to and side-effect analyses for Java. In: International Symposium on Software testing and analysis, pp. 1–11 (2002)

Typing Linear Constraints for Moding CLP($\mathcal{R}$) Programs

Salvatore Ruggieri[1] and Fred Mesnard[2]

[1] Dipartimento di Informatica, Università di Pisa, Italy
ruggieri@di.unipi.it
[2] Iremia, Université de la Réunion, France
frederic.mesnard@univ-reunion.fr

Abstract. We present a type system for linear constraints over reals and its use in mode analysis of CLP programs. The type system is designed to reason about the properties of definiteness, lower and upper bounds of variables of a linear constraint. Two proof procedures are presented for checking validity of type assertions. The first one considers lower and upper bound types, and it relies on solving homogeneous linear programming problems. The second procedure, which deals with definiteness as well, relies on computing the Minkowski's form of a parameterized polyhedron. The two procedures are sound and complete. We extend the approach to deal with strict inequalities and disequalities. Type assertions are at the basis of moding constraint logic programs. We extend the notion of well-moding from pure logic programming to CLP($\mathcal{R}$).

Keywords: linear constraints, polyhedra, constraint logic programming, well-moding, definiteness.

1 Introduction

Modes in logic programming allow the user to specify the input-output behaviour of predicate arguments [1]. Modern constraint logic programming languages adopt moding both as program annotation and as a tool for compiler optimizations, program transformations and termination analysis. As an example, consider the MORTGAGE program over CLP($\mathcal{R}$).

```
(m1)  mortgage(P,T,R,B) ←        (m2)  mortgage(P,T,R,B) ←
          T = 0,                             T >= 1,
          B = P.                             NP = P + P * 0.05 - R,
                                             NT = T - 1,
                                             mortgage(NP,NT,R,B).
```

The query ← `mortgage(100, 5, 20, B)` is intended to calculate the balance of a mortgage of 100 units after giving back 20 units per year for a period of 5 years. The answer provides an exact value (i.e., a real number) for the required balance, namely `B = 17.12`. Using the moding terminology, we say that given *definite* values for principal, time and repayment, in every answer we

© Springer-Verlag Berlin Heidelberg 2008

obtain a definite value for the balance. However, this is only one mode we can query the program above. The query $\leftarrow$ `3 <= T, T <= 5, mortgage(100, T, 20, B)` is intended to calculate the balance at the end of the third, fourth and fifth year. Principal and repayment are now definite, whilst time is (upper and lower) *bounded*. Again, for every answer we will get a definite value for balance. Intuitively, this mode is more general than the previous one, since definiteness of time has been replaced by boundedness. Finally, consider the query $\leftarrow$ `0 <= B, B <= 10, 15 <= R, R <= 20, mortgage(P, 5, R, B)`, which is intended to calculate the principal one could be granted such that by repaying from 15 to 20 units per year, after 5 years the balance yield is up to 10 units. The answer is now `P=0.78*B+4.33*R`, which is not definite, but, since `B` and `R` are bounded, it is (upper and lower) bounded. This mode is not comparable to the previous ones, since we now provide a definite value for time and a range for balance and repayment, and we wish to compute a range for the principal of the answer. These examples give only a few hints about the flexibility of the constraint logic programming scheme, even if compared to pure logic programming, where definiteness of variables corresponds to groundness, but upper and lower bounds have no direct equivalent.

In this paper, we concentrate on constraint languages with linear constraints over reals and rationals, as in CLP($\mathcal{R}$) [11], ECLiPSe, Sictus Prolog, SWI Prolog, and many others. We present a type system for linear constraints, where types model definiteness, upper and lower bounds of variables. Type assertions are introduced in order to derive types implied by a constraint and a set of typed variables. Validity of type assertions is thoroughly investigated by devising two proof procedures. The first one considers lower and upper bound types, and it relies on solving homogeneous linear programming problems. The second procedure, which deals with definiteness as well, relies on computing the Minkowski's form of a parameterized polyhedron. The two procedures are sound and complete. Moreover, the approach is extended to deal with constraints containing strict inequalities and disequalities. Type assertions are at the basis of moding CLP($\mathcal{R}$) programs. We extend the notion of well-moding from pure logic programming to CLP($\mathcal{R}$), showing useful properties in support of static analysis.

Preliminaries. We adhere to standard notation for linear algebra [16], linear programming [15] and (constraint) logic programming [1,10].

Linear Algebra. Small capital letters ($\mathbf{a}$, $\mathbf{b}$, ...) denote column vectors, while capital letters ($\mathbf{A}$, $\mathbf{B}$, ...) denote matrices. $\mathbf{0}$ and $\mathbf{1}$ are column vectors with all elements equal to 0 and 1 respectively. $\mathbf{a}_i$ denotes the i^{th} element in $\mathbf{a}$, and $row(\mathbf{A}, i)$ the row vector consisting of the i^{th} row of $\mathbf{A}$. $\mathbf{a}^T$ denotes the transposed vector of $\mathbf{a}$. $\mathbf{c}^T\mathbf{x}$ denotes the inner product of the transposed vector $\mathbf{c}^T$ and $\mathbf{x}$. $\Sigma\mathbf{v}$ is the sum of all the elements in $\mathbf{v}$. $\mathbf{A}\mathbf{x} \leq \mathbf{b}$ denotes a system of linear inequalities (or, a linear system) over the variables in $\mathbf{x}$. We assume that the dimensions of vectors and matrices in inner products and linear systems are of the appropriate size. The solution set of points that satisfy a formula/linear system ψ over $\mathcal{R}^n$ is defined as $Sol(\psi) = \{\mathbf{x} \in \mathcal{R}^n \mid \psi(\mathbf{x})\}$. A polyhedron is the solution set of a linear system, namely $Sol(\mathbf{Ax} \leq \mathbf{b})$.

Linear Programming. A linear programming problem consists of determining $max\{\mathbf{c}^T\mathbf{x} \mid \mathbf{Ax} \leq \mathbf{b}\}$, if it exists. The problem is infeasible when $\{\mathbf{x} \mid \mathbf{Ax} \leq \mathbf{b}\} = \emptyset$. If feasible, but $\{\mathbf{c}^T\mathbf{x} \mid \mathbf{Ax} \leq \mathbf{b}\}$ has no upper bound, the problem is unbounded, and we write $max\{\mathbf{c}^T\mathbf{x} \mid \mathbf{Ax} \leq \mathbf{b}\} = \infty$. Otherwise, it is bounded. We write $max\{\mathbf{c}^T\mathbf{x} \mid \mathbf{Ax} \leq \mathbf{b}\} \in \mathcal{R}$ when the problem is feasible and bounded. We extend the notation to a closed set of points S by writing $max\{\mathbf{c}^T\mathbf{x} \mid \mathbf{x} \in S\}$.

Constraint Logic Programming. The CLP Scheme defines a family of languages, CLP($\mathcal{C}$), that are parametric in the constraint domain $\mathcal{C}$. We are interested here in constraint domains over reals, such as CLP($\mathcal{R}$) [11]. All results apply to rationals as well. A primitive linear constraint is an expression $a_1 \cdot x_1 + \ldots a_n \cdot x_n \simeq a_0$, where $\simeq$ is in $\{\leq, =, \geq\}$, $a_1, \ldots, a_n$ are constants in $\mathcal{R}$ and $x_1, \ldots, x_n$ are variables. We will use the inner product form by rewriting it as $\mathbf{c}^T\mathbf{x} \simeq \alpha$. A linear constraint c is a sequence of primitive constraints, whose interpretation is their conjunction. A constraint logic program is a finite set of clauses of the form $A \leftarrow c, B_1, \ldots, B_n$, where A is an atom, c a linear constraint, and $B_1, \ldots, B_n$ $(n \geq 0)$ a sequence of atoms. We assume that atoms are in flat form, namely an atom is $p(x_1, \ldots, x_n)$ where p is a predicate of arity n and $x_1, \ldots, x_n$ are (not necessarily distinct) variables. A query $\leftarrow c, B_1, \ldots, B_n$ consists of a linear constraint and a sequence of atoms.

2 Bound Types for Linear Constraints

2.1 Syntax and Semantics

We introduce a static typing for variables in linear constraints. The set of types $\mathcal{BT}$ is defined first.

Definition 1 (types). *A* type *is an element of* $\mathcal{BT} = \{\star, \sqcup, \sqcap, \Box, !\}$.

The intuitive meaning of a type is to label variables occurring in a constraint on the basis of the values that they can assume in the set of solutions of the constraint. ! is intended to type variables that show at most one single value in every solution, a property known as *definiteness*; $\Box$ is intended to type variables that assume a range of values (hence, lower and upper bounds exist); $\sqcup$ (resp., $\sqcap$) is intended for variables that have a lower bound (resp., an upper bound); and finally, $\star$ is to be used when no upper or lower bound can be stated.

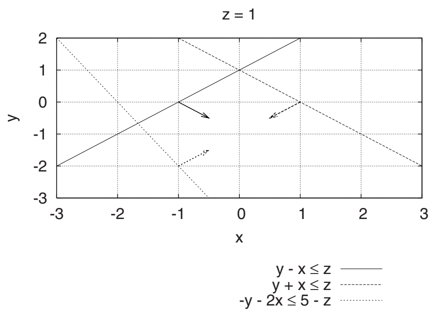
Let us introduce syntactic means to assert the type of variables.

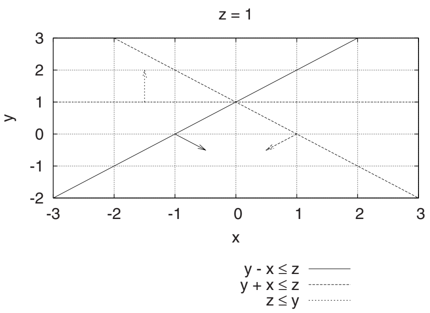
Definition 2 (types assertions). *An* atomic type declaration *(atd, for short) is an expression $x : \tau$, where x is a variable and $\tau \in \mathcal{BT}$. We define $vars(x : \tau) = \{x\}$, and say that x is typed as τ. A* type declaration *is a sequence of atd's $d_1, \ldots, d_n$, with $n \geq 0$. We define $vars(d_1, \ldots, d_n) = \cup_{i=1..n} vars(d_i)$.*

A type assertion *is an expression $\mathbf{d}_1 \vdash c \rightarrow \mathbf{d}_2$, where $\mathbf{d}_1, \mathbf{d}_2$ are type declarations and c is a linear constraint.*

Type declarations type variables. Such a typing is used in type assertions as an hypothesis (at the left of $\vdash$) or as a conclusion (at the right of $\rightarrow$). Intuitively, the type assertion $\mathbf{d}_1 \vdash c \rightarrow \mathbf{d}_2$ states that given the type declaration $\mathbf{d}_1$, the type declaration $\mathbf{d}_2$ holds under the linear constraint c.

Example 1. The type assertion $z :! \vdash y - x \leq z, y + x \leq z, -y - 2x \leq 5 - z \rightarrow y : \sqcap, x : \sqcup$ intuitively states that if z has a fixed value then the set of solutions of the involved constraint is such that y has an upper bound and x has a lower bound. The figure below (left) shows graphically the set of solutions for $z = 1$.





The type assertion $z :! \vdash y - x \leq z, y + x \leq z, z \leq y \rightarrow y :!, x :!$ states that if z has a fixed value then either the set of solutions of the involved constraint is empty or both x and y assume a unique value in it. The figure above (right) shows graphically the set of solutions for $z = 1$.

For a type declaration $\mathbf{d}$, we write $\mathbf{d}|_{\mathbf{x}}$ (resp., $\mathbf{d}|_{\tau}$) to denote the subsequence of $\mathbf{d}$ consisting only of atd's typing variables in $\mathbf{x}$ (resp., as τ). The intuition on the meaning of type assertions is formalized by the next definition.

Definition 3 (semantics). *We associate to an atd $d = x : \tau$ a formula $\phi(d)$ over fresh variables $\upsilon(d)$, called parameters, as follows:*

$$\begin{aligned}
\phi(x :!) &= x = a & \upsilon(x :!) &= \{a\}\\
\phi(x : \Box) &= a \leq x \wedge x \leq b & \upsilon(x : \Box) &= \{a, b\}\\
\phi(x : \sqcup) &= a \leq x & \upsilon(x : \sqcup) &= \{a\}\\
\phi(x : \sqcap) &= x \leq b & \upsilon(x : \sqcap) &= \{b\}\\
\phi(x : \star) &= \mathtt{true} & \upsilon(x : \star) &= \emptyset.
\end{aligned}$$

ϕ and υ extend to type declarations as follows:

$$\phi(d_1, \ldots, d_n) = \wedge_{i=1..n} \phi(d_i) \qquad \upsilon(d_1, \ldots, d_n) = \cup_{i=1..n} \upsilon(d_i).$$

A type assertion $\mathbf{d}_1 \vdash c \rightarrow \mathbf{d}_2$ is valid if for $\mathbf{v} = vars(c) \cup vars(\mathbf{d}_1) \cup vars(\mathbf{d}_2)$, the following formula is true in the domain of reals:

$$\forall \upsilon(\mathbf{d}_1) \exists \upsilon(\mathbf{d}_2) \forall \mathbf{v}.(\phi(\mathbf{d}_1) \wedge c) \rightarrow \phi(\mathbf{d}_2). \tag{1}$$

Example 2. For the type assertion $z :! \vdash y - x \leq z, y + x \leq z, z \leq y \rightarrow y :!, x :!$, the formula to be shown is:

$$\forall a \; \exists b, c \; \forall x, y, z. \; (z = a \wedge y - x \leq z \wedge y + x \leq z \wedge z \leq y) \rightarrow (y = b \wedge x = c).$$

The set of variables is fixed to $\mathbf{v} = vars(c) \cup vars(\mathbf{d}_1) \cup vars(\mathbf{d}_2)$ in order to take into account variables that appear in $\mathbf{d}_1$ or $\mathbf{d}_2$ but not in c, e.g. in the (valid) type assertion $x : \sqcap \vdash \texttt{true} \rightarrow x : \sqcap$.

A natural ordering over types is induced by the semantics above. For instance, it is readily checked that $\mathbf{d} \vdash c \rightarrow x :!$ implies $\mathbf{d} \vdash c \rightarrow x : \Box$ for any $\mathbf{d}$, c and x. Similar implications lead to define an order $\geq_t$ over types.

Definition 4. *The $\geq_t$ partial order over $\mathcal{BT}$ is defined as the reflexive and transitive closure of the following relation $\rightarrow$:*

$$\begin{tikzcd}
 & & \sqcap \arrow[dr] & \\
! \arrow[r] & \Box \arrow[ur] \arrow[dr] & & \star \\
 & & \sqcup \arrow[ur] &
\end{tikzcd}$$

We write $\tau >_t \mu$ when $\tau \geq_t \mu$ and $\tau \neq \mu$. We define $lub(\emptyset) = \star$ and for $n > 0$:

$$lub(\{\tau_1, \ldots, \tau_n\}) = min\{\tau \mid \tau \geq_t \tau_i, i = 1..n\}.$$

Next, the $\geq_t$ relation is extended to type declarations.

Definition 5. *We write $\mathbf{d}_1 \geq_t \mathbf{d}_2$ if for every $x : \tau$ in $\mathbf{d}_2$ there exists $x : \mu$ in $\mathbf{d}_1$ such that $\mu \geq_t \tau$.*

Using the notation $\geq_t$, the intuition behind the ordering can be formalized by a monotonicity lemma. Also, transitivity is readily checked.

Lemma 1 (monotonicity). *Assume that $\mathbf{d}_1 \vdash c \rightarrow \mathbf{d}_2$ is valid. If $\mathbf{d}'_1 \geq_t \mathbf{d}_1$, $\mathbf{d}_2 \geq_t \mathbf{d}'_2$ and $\mathcal{R} \models c' \rightarrow c$ for a linear constraint c', then $\mathbf{d}'_1 \vdash c' \rightarrow \mathbf{d}'_2$ is valid.*

Normal forms for type declarations are introduced by assigning to each variable the least upper bound of its types. When the least upper bound is $\star$, the type assignment provides no actual information and then it can be discarded. Normal forms are unique modulo reordering of atd's.

Definition 6. *We define $nf(\mathbf{d})$ as any type declaration $\mathbf{d}'$ such that $x : \tau$ is in $\mathbf{d}'$ iff $\tau = lub(\{\mu \mid x : \mu \text{ is in } \mathbf{d}\ \})$ and $\tau \neq \star$.*

Example 3. Notice that $x : \Box \geq_t x : \sqcap, x : \sqcup$ holds, while $x : \sqcap, x : \sqcup \geq_t x : \Box$ does not hold. Actually, $\geq_t$ does not capture semantic implication. We have to move to normal forms to conclude that $nf(x : \sqcap, x : \sqcup) = x : \Box \geq_t x : \Box$.

Normal forms precisely characterize validity when it only depends on type declarations, i.e. for the constraint $\texttt{true}$.

Lemma 2. $\mathbf{d}_1 \vdash \mathtt{true} \rightarrow \mathbf{d}_2$ *is valid iff* $nf(\mathbf{d}_1) \geq_t nf(\mathbf{d}_2)$.

2.2 Checking Type Assertions: First Intuitions

In principle, formulas as in (1) can be checked by real quantifier elimination methods [6], which trace back to Tarski's decision procedure for first order formula over real polynomials. However, while quantifier elimination represents a direct solution to the checking problem and it allows for generalizing to the non-linear case, we observe that formulas in (1) represent a quite restricted class. We will be looking for a specialized and efficient approach to check them. In addition, we are interested in the problem of inferring the largest (w.r.t. the $\geq_t$ order) $\mathbf{d}'$ such that $\mathbf{d} \vdash c \rightarrow \mathbf{d}'$ is valid, given $\mathbf{d}$ and c. Our approach switches from the *logical* view of constraints-as-formulas to a *geometric* view of constraints-as-polyhedra. Consider a linear constraint c and a type declaration $\mathbf{d}$. We observe that c can be equivalently represented as a linear system of inequalities $\mathbf{A}_c\mathbf{v} \leq \mathbf{b}_c$ where $\mathbf{v} = vars(c) \cup vars(\mathbf{d})$. The set of solutions of c coincides then with the polyhedron represented by $\mathbf{A}_c\mathbf{v} \leq \mathbf{b}_c$, which we call the *geometric* representation of c. Analogously, the linear constraint $\phi(\mathbf{d})$ can be represented as $\mathbf{A}_\mathbf{d}\mathbf{v} \leq \mathbf{B}_\mathbf{d}\mathbf{a}_\mathbf{d}$, where $\mathbf{a}_\mathbf{d}$ is the symbolic vector of parameters in $\upsilon(\mathbf{d})$. The resulting system $\phi(\mathbf{d}) \wedge c$ is a parameterized system of linear inequalities $\mathcal{P}$, where variables in $\upsilon(\mathbf{d})$ play the role of parameters:

$$\begin{pmatrix} \mathbf{A}_c \\ \mathbf{A}_\mathbf{d} \end{pmatrix} \mathbf{v} \leq \begin{pmatrix} \mathbf{b}_c \\ \mathbf{0} \end{pmatrix} + \begin{pmatrix} \mathbf{0} \\ \mathbf{B}_\mathbf{d} \end{pmatrix} \mathbf{a}_\mathbf{d} \tag{2}$$

The notion of parameterized polyhedra models the solutions of parameterized linear systems.

Definition 7 (Parameterized polyhedron). *A* parameterized polyhedron *is a collection of polyhedra defined by fixing the value for parameters in a parameterized system of linear inequalities:* $Sol(\mathbf{Ax} \leq \mathbf{b}+\mathbf{Ba}, \mathbf{u}) = \{\mathbf{x} \mid \mathbf{Ax} \leq \mathbf{b}+\mathbf{Bu}\}$.

$Sol()$ is now a binary function. In addition to a system of parameterized linear inequalities, an assignment to parameters is required.

Example 4. Let $\mathbf{d}$ be z :! and c be $y - x \leq z, y + x \leq z, -y - 2x \leq 5 - z$. We have that $\phi(\mathbf{d})$ is $z = a$, and then the parameterized system for $\phi(\mathbf{d}) \wedge c$ is:

$$\begin{pmatrix} -1 & 1 & -1 \\ 1 & 1 & -1 \\ -2 & -1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} \mathrm{x} \\ \mathrm{y} \\ \mathrm{z} \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ 5 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ -1 \end{pmatrix} a$$

Under this interpretation, validity of $\mathbf{d} \vdash c \rightarrow x : \tau$ has an intuitive geometric interpretation. Assume that $x = \mathbf{v}_i$. $\mathbf{d} \vdash c \rightarrow x : \tau$ is valid iff for every $\mathbf{u} \in \mathcal{R}^{|\upsilon(\mathbf{d})|}$, the set of solution points $\mathcal{S}_\mathbf{u} = Sol(\mathcal{P}, \mathbf{u})$ either is empty or:

- if $\tau = !$ then $max\{\mathbf{v}_i \mid \mathbf{v} \in \mathcal{S}_{\mathbf{u}}\} = min\{\mathbf{v}_i \mid \mathbf{v} \in \mathcal{S}_{\mathbf{u}}\} \in \mathcal{R}$, namely x assumes a single value;
- if $\tau = \square$ then $max\{\mathbf{v}_i \mid \mathbf{v} \in \mathcal{S}_{\mathbf{u}}\} \in \mathcal{R}$ and $min\{\mathbf{v}_i \mid \mathbf{v} \in \mathcal{S}_{\mathbf{u}}\} \in \mathcal{R}$ namely both an upper and a lower bound exist for x;
- if $\tau = \sqcup$ then $min\{\mathbf{v}_i \mid \mathbf{v} \in \mathcal{S}_{\mathbf{u}}\} \in \mathcal{R}$, namely a lower bound exists for x;
- if $\tau = \sqcap$ then $max\{\mathbf{v}_i \mid \mathbf{v} \in \mathcal{S}_{\mathbf{u}}\} \in \mathcal{R}$, namely an upper bound exists for x;
- if $\tau = \star$ then we have nothing to show.

Unfortunately, this procedure is not effective, since there are infinitely many $\mathcal{S}_{\mathbf{u}}$ to be checked. In the next two subsections, we will develop approaches for turning the intuitions above into effective and efficient procedures.

2.3 Checking Type Assertions: An LP Approach

In this section we develop an inference algorithm which does not explicitly take into account parameters. We will be able to reason on type assertions over $\mathcal{BT} \setminus \{!\}$. First of all, let us consider the case of unsatisfiable constraints.

Lemma 3. *Consider the parameterized polyhedron $\mathcal{P}$ in (2). There exists a parameter instance $\mathbf{u}$ such that $Sol(\mathcal{P}, \mathbf{u}) \neq \emptyset$ iff $Sol(\mathbf{A}_c\mathbf{v} \leq \mathbf{b}_c) \neq \emptyset$.*

As a consequence, if $Sol(\mathbf{A}_c\mathbf{v} \leq \mathbf{b}_c) = \emptyset$ (i.e., c is an unsatisfiable constraint) then there is no chance to obtain a non-empty polyhedron by some instantiation of the parameters in $\phi(\mathbf{d})$. In this case, we can infer assertions of the form $\mathbf{d} \vdash c \rightarrow x :!$, for every variable x. From now on, we will concentrate then on satisfiable constraints. As it will be recalled later on, a non-empty polyhedron $Sol(\mathbf{Ax} \leq \mathbf{b})$ can be decomposed into the vectorial sum of its characteristic cone $Sol(\mathbf{Ax} \leq \mathbf{0})$ with a polytope, a polyhedra bounded along every dimension. Therefore, the existence of an upper/lower bound for a linear function over a polyhedron depends only on its characteristic cone. It is immediate to observe that for every parameter instance $\mathbf{u}$, the polyhedra $Sol(\mathcal{P}, \mathbf{u})$ share the same characteristic cone. As a consequence, proving the existence of an upper bound is independent from the parameter instance, and it relies only on the homogeneous version of $\mathcal{P}$, which is not anymore parameterized.

Lemma 4. *Consider the parameterized polyhedron $\mathcal{P}$ in (2). Let $\mathcal{H}$ be its homogeneous version: $\mathbf{A}_c\mathbf{v} \leq \mathbf{0}$, $\mathbf{A}_{\mathbf{d}}\mathbf{v} \leq \mathbf{0}$, and assume that $Sol(\mathbf{A}_c\mathbf{v} \leq \mathbf{b}_c) \neq \emptyset$.*

We have that $max\{\mathbf{c}^T\mathbf{v} \mid \mathbf{v} \in Sol(\mathcal{H})\} = 0$ iff for every parameter instance $\mathbf{u}$, $Sol(\mathcal{P}, \mathbf{u}) = \emptyset$ or $max\{\mathbf{c}^T\mathbf{v} \mid \mathbf{v} \in Sol(\mathcal{P}, \mathbf{u})\} \in \mathcal{R}$.

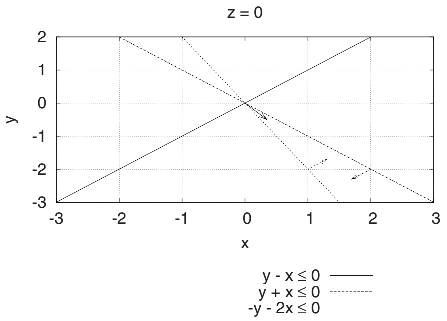
When $\mathbf{c}$ is always 0 except for the i^{th} position where it is 1, we have $\mathbf{c}^T\mathbf{v} = \mathbf{v}_i$. Lemma 4 solves then the problem of deciding whether $\mathbf{d} \vdash c \rightarrow \mathbf{v}_i : \sqcap$, without having to take into account parameters. By reasoning similarly for types $\sqcup$ and $\square$, we can state an effective procedure, called LPCheck and summarized in Fig. 1, which outputs type declarations in normal form without any trivial atd.

Example 5. The homogeneous version of the parameterized linear system in Example 4 and its graphical representation are the following:

```
Input: a type assertion $\mathbf{d}_1$, a linear constraint $c$ and a sequence of variables $\mathbf{x}$.

Step 0 Define $\mathbf{v} = vars(c)$, $\mathbf{n} = nf(\mathbf{d}_1)$, $\mathbf{d} = \mathbf{n}|_{\mathbf{v}}$.
Step 1 Let $\mathbf{A}_c\mathbf{v} \leq \mathbf{b}_c$ be the geometric representation of $c$, and $\mathbf{A_d}\mathbf{v} \leq \mathbf{B_d}\mathbf{a_d}$ the
    geometric representation of $\phi(\mathbf{d})$.
Step 2 If $Sol(\mathbf{A}_c\mathbf{v} \leq \mathbf{b}_c) = \emptyset$ Then for every $x$ in $\mathbf{x}$, output "$x$ :!"
       Else
Step 3   for every $x$ in $\mathbf{x} \setminus \mathbf{v}$ and $x : \tau$ in $\mathbf{n}$, output "$x : \tau$";
Step 4   for every $x$ in $\mathbf{x} \cap \mathbf{v}$:
    (a) Let $M = max\{x \mid \mathbf{A}_c\mathbf{v} \leq \mathbf{0}, \mathbf{A_d}\mathbf{v} \leq \mathbf{0}\}$ and $m = max\{-x \mid \mathbf{A}_c\mathbf{v} \leq \mathbf{0}, \mathbf{A_d}\mathbf{v} \leq \mathbf{0}\}$.
    (b) Output "$x : \Box$" if $M = 0$ and $m = 0$;
    (c) Output "$x : \sqcup$" if $M = \infty$ and $m = 0$;
    (d) Output "$x : \sqcap$" if $M = 0$ and $m = \infty$.
```

Fig. 1. LPCheck procedure

$$\begin{pmatrix} -1 & 1 & -1 \\ 1 & 1 & -1 \\ -2 & -1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$



It is readily checked that x has a lower bound and y has an upper bound.

Soundness and a relative form of completeness of procedure LPCheck follow.

Theorem 1 (LPCheck - soundness and completeness). *Let $\mathbf{d}_1$ be a type declaration and c a linear constraint. If the sequence $\mathbf{d}_2$ is provided as output by* LPCheck*, the type assertion $\mathbf{d}_1 \vdash c \rightarrow \mathbf{d}_2$ is valid. Conversely, assume that $\mathbf{d}_1 \vdash c \rightarrow \mathbf{d}_2$ is valid, and that c is unsatisfiable or no variable in $vars(c)$ is typed as ! in $\mathbf{d}_2$. Then there exists a sequence $\mathbf{d}$ provided as output by* LPCheck *such that $\mathbf{d} \geq_t nf(\mathbf{d}_2)$.*

The LPCheck procedure is not tied to any underlying linear programming solver. By adopting a polynomial time algorithm [15,16], we can conclude that LPCheck has a polynomial time complexity. Due to the approach that we will follow later on for dealing with parameters, we present here an instantiation of LPCheck which consists of directly computing the generating matrix and the vertex matrix of polyhedra. This is an alternative representation of polyhedra, known as the explicit representation or the Minkowski's form [16, Section 8.9].

Theorem 2 (Minkowski's decomposition theorem for polyhedra). *There exists an effective procedure that given $\mathbf{Ax} \leq \mathbf{b}$ decides whether or not the* *polyhedron* $Sol(\mathbf{Ax} \leq \mathbf{b})$ *is empty and, if not, it yields a generating matrix* $\mathbf{R}$ *and a vertex matrix* $\mathbf{V}$ *such that:*

$$Sol(\mathbf{Ax} \leq \mathbf{b}) = \{\mathbf{x} \mid \mathbf{x} = \mathbf{R}\boldsymbol{\lambda}, \boldsymbol{\lambda} \geq 0 \} + \{\mathbf{x} \mid \mathbf{x} = \mathbf{V}\boldsymbol{\gamma}, \boldsymbol{\gamma} \geq 0, \Sigma\boldsymbol{\gamma} = 1 \},$$

and $Sol(\mathbf{Ax} \leq \mathbf{0}) = \{\mathbf{x} \mid \mathbf{x} = \mathbf{R}\boldsymbol{\lambda}, \boldsymbol{\lambda} \geq 0 \}$.

A column of $\mathbf{R}$ is called a *ray*: for any $\mathbf{x}_0 \in Sol(\mathbf{Ax} \leq \mathbf{b})$ and ray $\mathbf{r}$, it turns out that $\mathbf{r}\lambda + \mathbf{x}_0 \in Sol(\mathbf{Ax} \leq \mathbf{b})$ for every $\lambda \geq 0$. A column of $\mathbf{V}$ is called a *vertex*. The set $ConvexHull(\mathbf{V}) = \{\mathbf{x} \mid \mathbf{x} = \mathbf{V}\boldsymbol{\gamma}, \gamma \geq 0, \Sigma\gamma = 1 \}$, where $\mathbf{V}$ is a matrix or a finite set of vectors, is the convex hull of the vertices, namely the smallest convex set which contains all vertices. An efficient procedure to extract minimal $\mathbf{R}$ and $\mathbf{V}$ is the double description method, also known as the Motzkin-Chernikova-Le Verge algorithm [3,17]. Turning on the LPCheck procedure, the satisfiability test at **Step 2** is performed as part of the construction of the explicit representation of the polyhedron. The maximization problems at **Step 4 (a)** can easily be solved directly on the explicit representation.

Lemma 5. *Consider a characteristic cone* $Sol(\mathbf{Ax} \leq \mathbf{0})$*, and let* $\mathbf{R}$ *be its generating matrix. We have that* $max\{\mathbf{c}^T\mathbf{x} \mid \mathbf{Ax} \leq \mathbf{0}\} = 0$ *iff* $\mathbf{c}^T\mathbf{R} \leq \mathbf{0}$.

Since in our context $\mathbf{c}$ is always zero except for the i^{th} element, which is 1 or -1, we can conclude that a variable $\mathbf{v}_i$ (the i^{th} variable in $\mathbf{v}$) is bounded from above by 0 (resp., bounded from below by 0) iff all values in $row(\mathbf{R}, i)$ are non-positive (resp., non-negative).

Example 6. The Minkowski's form of the homogeneous system in Example 5 is:

$$\begin{pmatrix} 1 & 1 \\ -2 & -1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix}, \lambda_1 \geq 0, \lambda_2 \geq 0$$

Intuitively, the two columns in the generating matrix $\mathbf{R}$ correspond to vectors lying on the two borders of the cone in the graph of Example 5. Using Lemma 5, it is readily checked that when $\mathbf{c}$ is one of $(-1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$ or $(0, 0, -1)$ then $\mathbf{c}^T\mathbf{R} \leq \mathbf{0}$, i.e. x is bounded from below, y from above, and z from both.

2.4 Checking Type Assertions: A Parameterized Approach

In this section, we reason on the parameterized system in (2) by adopting an approach that explicitly considers parameters [14], and which is an extension of the Minkowski's decomposition theorem. However, the complexity of the approach in presence of k parameters is polynomially proportional (by a k^3 factor) to its complexity in absence of parameters. For this reason, we first ask ourselves whether we can build on the results of the last subsection. The LPCheck procedure is sound, and it is also complete except for the ! type. Thus, we are restricted with checking assertions of the form $\mathbf{d} \vdash c \rightarrow x :!$. Under this context, are all typings in $\mathbf{d}$ necessary?

Example 7. Consider $x : \Box \vdash z = x, z - y = 2, z + y = 0 \rightarrow x :!, y :!, z :!$. Starting from the involved constraint, by Gaussian elimination, we derive: $x = z, y = z - 2$, $2z = 2$ and then $z = 1, y = -1, x = 1$. Hence the type assertion is valid.

Notice that we made no use of $x : \Box$ in proving validity of the type assertion. This fact can be generalized, in order to get rid of unnecessary parameters.

Theorem 3 (Definiteness). $\mathbf{d} \vdash c \rightarrow x :!$ *is valid iff* $\mathbf{d}|_! \vdash c \rightarrow x :!$ *is valid.*

Let us consider now an example which illustrates the Fourier-Motzkin elimination method for linear inequalities applied in presence of parameters.

Example 8. Consider the constraint c defined as $y + x \leq z, y - x \leq z, z \leq y, 0 \leq z, w \leq z$, and the type declaration $z :!$. We start by isolating variable y in $\phi(z :!) \wedge c$, as shown at **(a)** in the figure below.

$$\begin{array}{rcl}
 & y & \leq z - x \\
 & y & \leq z + x \\
z \leq & y & \\
0 \leq & z & \\
 & w & \leq z \\
 & z = a & \\
 & \textbf{(a)} &
\end{array}
\qquad\qquad
\begin{array}{rcl}
z \leq & y & \leq min\{z - x, z + x\} \\
 & x = 0 & \\
0 \leq & z & \\
 & w & \leq z \\
 & z = a & \\
 & \textbf{(b)} &
\end{array}
\qquad\qquad
\begin{array}{rcl}
 & y = a & \\
 & x = 0 & \\
 & w & \leq a \\
 & z = a & \\
0 \leq & a & \\
 & \textbf{(c)} &
\end{array}$$

Bounds for variable y can then be summarized as: $(*)\ z \leq y \leq min\{z - x, z + x\}$. Moreover, the bounds $e_1 \leq e_2$ are implied for e_1 expression bounding y from below in $(*)$, and e_2 bounding y from above in $(*)$. Actually, the original set of linear inequalities over y is equivalent to $z \leq y \leq min\{z - x, z + x\}$ plus such bounds. The new inequality set is reported at **(b)** in the figure above. By replacing backward $x = 0$ and $z = a$, we end up with the final system at **(c)** in the figure above, where no further elimination is possible. The final system is feasible when the condition $0 \leq a$ holds. In this system, we have $x = 0, y = a, z = a$. Moreover, $w \leq a$ can be rewritten as: $w = -\lambda_1 + a$ for some $\lambda_1 \geq 0$. Put in a geometrical form, the solution set of $\phi(z :!) \wedge c$ is:

$$\left\{ (x, y, z, w) \mid \begin{array}{c} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ -1 \end{pmatrix} \left(\lambda_1 \right) + \begin{pmatrix} 0 \\ a \\ a \\ a \end{pmatrix} \\ \text{for every } \lambda_1 \geq 0, \quad \text{when } a \geq 0 \end{array} \right\}$$

Summarizing, the values of x, y and z are univocally determined once the parameter a has been fixed and the system is feasible. Under the same hypotheses, the value of w is bounded from above (by a), but it is not definite. With our notation, $z :! \vdash c \rightarrow x :!, y :!, z :!, w : \sqcap$ is valid.

The final form reached at the end of the example resembles the Minkowski's form for polyhedra, but with a parameterized vector appearing in the vertex matrix.

```
Input: a type assertion $\mathbf{d}_1$, a linear constraint $c$ and a sequence of variables $\mathbf{x}$.

Step 0 Define $\mathbf{v} = vars(c)$, $\mathbf{n} = nf(\mathbf{d}_1)$, $\mathbf{d} = \mathbf{n}|_!$.
Step 1 Let $\mathbf{A}_c\mathbf{v} \leq \mathbf{b}$ be the geometric representation of $c$, and $\mathbf{A_d v} \leq \mathbf{B_d a_d}$ the
    geometric representation of $\phi(\mathbf{d})$.
Step 1 For the parametric polyhedron $\begin{pmatrix}\mathbf{A}_c \\ \mathbf{A_d}\end{pmatrix}\mathbf{v} \leq \begin{pmatrix}\mathbf{b}_c \\ \mathbf{0}\end{pmatrix} + \begin{pmatrix}\mathbf{0} \\ \mathbf{B_d}\end{pmatrix}\mathbf{a}$, build the gen-
    erating matrix $\mathbf{R}$ and the sequence $(\mathbf{v^a}(1), \mathbf{C}_1\mathbf{a} \leq \mathbf{c}_1), \ldots, (\mathbf{v^a}(k), \mathbf{C}_k\mathbf{a} \leq \mathbf{c}_k)$
Step 2 For every $x : \tau$ as output from LPCheck
Step 3  If $\tau \neq \square$ or $x \notin \mathbf{v}$ Then output "$x : \tau$";
Step 4  Else let $i$ such that $x = \mathbf{v}_i$:
        (a) Output "$x$ :!" if $row(\mathbf{R}, i) = \mathbf{0}$ and for $1 \leq m < n \leq k$, $\mathbf{v^a}(m)_i = \mathbf{v^a}(n)_i$
            over $\mathbf{C}_m\mathbf{a} \leq \mathbf{c}_m$, $\mathbf{C}_n\mathbf{a} \leq \mathbf{c}_n$;
        (b) Output "$x : \square$" otherwise.
```

Fig. 2. POLYCheck procedure

The generalization of the Minkowski's theorem to parameterized polyhedra is provided in [14] and implemented in the `polylib` library [13].

Theorem 4 (Minkowski's theorem for parameterized polyhedra). *Every parameterized polyhedron can be expressed by a generating matrix* $\mathbf{R}$ *and finitely many pairs* $(\mathbf{v^a}(1), \mathbf{C}_1\mathbf{a} \leq \mathbf{c}_1), \ldots, (\mathbf{v^a}(k), \mathbf{C}_k\mathbf{a} \leq \mathbf{c}_k)$ *where, for* $i = 1..k$, $\mathbf{v^a}(i)$ *is a vector parametric in* $\mathbf{a}$ *and* $Sol(\mathbf{C}_i\mathbf{a} \leq \mathbf{c}_i) \neq \emptyset$*, as follows:*

$$Sol(\mathbf{Ax} \leq \mathbf{b} + \mathbf{Ba}, \mathbf{u}) = \{\mathbf{x} \mid \mathbf{x} = \mathbf{R}\boldsymbol{\lambda}, \boldsymbol{\lambda} \geq 0\} + ConvexHull(\{\mathbf{v^u}(i) \mid i = 1..k, \mathbf{C}_i\mathbf{u} \leq \mathbf{c}_i\}),$$

and $Sol(\mathbf{Ax} \leq \mathbf{0}) = \{\mathbf{x} \mid \mathbf{x} = \mathbf{R}\boldsymbol{\lambda}, \boldsymbol{\lambda} \geq 0\}$.

The vertex matrix is now replaced by a set of pairs where the first element is a parameterized vertex and the second one is its *validity domain*. For a parameter instance $\mathbf{u}$, the vertex matrix is built from the (instantiated) vertices whose validity domain includes $\mathbf{u}$. The special case $k = 0$ models *empty parameterized polyhedra*, which are empty for any instance of the parameters. Now that we have an explicit form for parameterized polyhedra, we need a procedure to test whether a variable is definite for every parameter instance. First, we introduce a notion to test whether two expressions are equal over a polyhedron.

Definition 8. *We say that* $\mathbf{c}_1^T\mathbf{x} + \alpha_1 = \mathbf{c}_2^T\mathbf{x} + \alpha_2$ *over* $\mathbf{Ax} \leq \mathbf{b}$ *if for every* $\mathbf{x}_0 \in Sol(\mathbf{Ax} \leq \mathbf{b})$, $\mathbf{c}_1^T\mathbf{x}_0 + \alpha_1 = \mathbf{c}_2^T\mathbf{x}_0 + \alpha_2$.

Given the Minkowski's form, equality can be checked as follows.

Lemma 6. $\mathbf{c}_1^T\mathbf{x} + \alpha_1 = \mathbf{c}_2^T\mathbf{x} + \alpha_2$ *over* $\mathbf{Ax} \leq \mathbf{b}$ *iff* $(\mathbf{c}_1 \;\; \alpha_1) = (\mathbf{c}_2 \;\; \alpha_2)$ *or* $Sol(\mathbf{Ax} \leq \mathbf{b}) = \emptyset$ *or, called* $\mathbf{R}$ *and* $\mathbf{V}$ *the generating and vertex matrices of* $\mathbf{Ax} \leq \mathbf{b}$, $(\mathbf{c}_1 - \mathbf{c}_2)^T\mathbf{R} = \mathbf{0}$ *and* $(\mathbf{c}_1 - \mathbf{c}_2)^T\mathbf{V} = (\alpha_2 - \alpha_1)\mathbf{1}^T$.

Notice that checking $(\mathbf{c}_1 \;\; \alpha_1) = (\mathbf{c}_2 \;\; \alpha_2)$ is not strictly necessary, but if we interpret the three conditions in the lemma from a computational point of view, the first one provides a very fast test. The next result states that definiteness of a variable x over a parameterized polyhedron amounts to showing that no pair of vertices is such that their projections over x differ for any parameter instance in the non-empty intersection of their domains.

Lemma 7. *Consider the Minkowski's form of a non-empty parameterized polyhedron as in Theorem 4. Every* $\mathcal{S}_{\mathbf{u}} = \{\mathbf{c}^T\mathbf{x} \mid \mathbf{x} \in Sol(\mathbf{Ax} \leq \mathbf{b}+\mathbf{Ba}, \mathbf{u})\}$ *is empty or a singleton iff* $\mathbf{c}^T\mathbf{R} = \mathbf{0}$ *and for* $1 \leq m < n \leq k$, $\mathbf{c}^T\mathbf{v}^{\mathbf{a}}(m) = \mathbf{c}^T\mathbf{v}^{\mathbf{a}}(n)$ *over* $\mathbf{C}_m\mathbf{a} \leq \mathbf{c}_m, \mathbf{C}_n\mathbf{a} \leq \mathbf{c}_n$.

Example 9. Consider a parameterized polyhedron over parameters $(a \;\; b)$ and variables $(x \;\; y)$ with generating matrix $\mathbf{0}$, and with pairs of vertices and domains:

$$(\begin{pmatrix} a \\ b \end{pmatrix}, b \geq a \geq 0) \qquad (\begin{pmatrix} a \\ a \end{pmatrix}, a \geq b \geq 0) \qquad (\begin{pmatrix} a \\ a+b \end{pmatrix}, a \geq 0 \wedge b \geq 0).$$

Let us reason about definiteness of variables x and y by using Lemma 7. x is definite, since $a = a$ over any polyhedron. Consider now y. For the first two vertices, we have $b \neq a$, or, in vectorial notation, $(0 \;\; 1)\begin{pmatrix} a \\ b \end{pmatrix} + \alpha_1 \neq (1 \;\; 0)\begin{pmatrix} a \\ b \end{pmatrix} + \alpha_2$ where $\alpha_1 = \alpha_2 = 0$. Since the intersection of the two domains, namely $a = b \geq 0$, is not empty, by Lemma 6 we proceed by computing its generating and vertex matrices. The vertex matrix is $\mathbf{0}$, so we simply have:

$$Sol(a = b \geq 0) = \{(a, b) \mid \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \left(\lambda_1 \right), \lambda_1 \geq 0\},$$

and we have to test: $(\left(0\,1\right) - \left(1\,0\right))\begin{pmatrix} 1 \\ 1 \end{pmatrix} = \left(0\right)$, where (0) is $(\alpha_2 - \alpha_1)\mathbf{1}^T$. Therefore, a and b are equal over $a = b \geq 0$. Consider now the first and the third vertex. Since $b \neq a+b$, we compute, as before, the Minkowski's form of the intersection of their domains:

$$Sol(b \geq a \geq 0) = \{(a, b) \mid \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 1\,0 \\ 1\,1 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix}, \lambda_1 \geq 0, \lambda_2 \geq 0\}.$$

Then we test: $(\left(0\,1\right) - \left(1\,1\right))\begin{pmatrix} 1\,0 \\ 1\,1 \end{pmatrix} = \left(-1\,0\right)$, which differs from the expected $\left(0\,0\right) = (\alpha_2 - \alpha_1)\mathbf{1}^T$ for $\alpha_2 = \alpha_1 = 0$. Summarizing, by Lemma 6, b and $a+b$ are not equal over $b \geq a \geq 0$, and then, by Lemma 7, y is not definite.

Lemmas 6 and 7 provide us with a checking procedure for definiteness. The overall procedure, called POLYCheck, is shown in Fig. 2. POLYCheck terminates and is sound and complete for inferring validity of type assertions.

Theorem 5 (POLYCheck - soundness and completeness). *Let* $\mathbf{d}_1$ *be a type declaration and* c *a linear constraint. If the sequence* $\mathbf{d}_2$ *is provided as output by* POLYCheck*, the type assertion* $\mathbf{d}_1 \vdash c \rightarrow \mathbf{d}_2$ *is valid. Conversely, assume that* $\mathbf{d}_1 \vdash c \rightarrow \mathbf{d}_2$ *is valid. Then there exists a sequence* $\mathbf{d}$ *provided as output by* POLYCheck *such that* $\mathbf{d} \geq_t nf(\mathbf{d}_2)$.

2.5 Extensions to Strict Inequalities and to Disequalities

So far, we considered equality and non-strict inequality primitive constraints. A generalized linear constraint admits primitive constraint over the operators $<$, $>$ (strict inequalities) and $\neq$ (disequalities). Without any loss of generality, we write a generalized constraint as $c \wedge \bigwedge_{i=1}^{m} e_i \neq \alpha_i$, where c is a linear constraint and for $i = 1..m$, $e_i \neq \alpha_i$ is a disequality. We now extend type assertions to admit generalized constraints. The next result shows that validity of type assertions for a satisfiable generalized constraint can be reduced to validity of the type assertions over the linear constraint obtained by removing the disequalities in it.

Theorem 6. *Let $g = c \wedge \bigwedge_{i=1}^{m} e_i \neq \alpha_i$ be a satisfiable generalized linear constraint. $\mathbf{d}_1 \vdash g \rightarrow \mathbf{d}_2$ is valid iff $\mathbf{d}_1 \vdash c \rightarrow \mathbf{d}_2$ is valid.*

Checking satisfiability of g is easily accomplished when computing the explicit form of polyhedra. By independence of negative constraints [12], it reduces to show that $Sol(\mathbf{A}_c\mathbf{v} \leq \mathbf{b}_c) \neq \emptyset$ and that every hyperplane $e_i = \alpha_i$ does not include the polyhedron $Sol(\mathbf{A}_c\mathbf{v} \leq \mathbf{b}_c)$, i.e., by Definition 8 that $e_i = \alpha_i$ over $\mathbf{A}_c\mathbf{v} \leq \mathbf{b}_c$ is false. Lemma 6 provides us with a procedure to check it.

3 Moding CLP Programs

Modes for pure logic programs assign to every predicate argument an input-output behavior. Input means that the predicate argument is ground on calls. Output means that it is ground on answers. As discussed in the introduction, groundness (i.e., definiteness) is restrictive in the CLP context. Based on types, we can extend the notion of moding to upper and/or lower bounds as well.

Definition 9 (moding). A mode *for a n-ary predicate p is a function d_p from $\{1, \ldots, n\}$ to $\mathcal{BT} \times \mathcal{BT}$. We write d_p as $p(\tau_1 \times \mu_1, \ldots, \tau_n \times \mu_n)$, where $d_p(i) = (\tau_i, \mu_i)$ for $i = 1..n$.*

A mode *for a CLP($\mathcal{R}$) program P is a set of modes, one for each predicate in P. For an atom $p(\mathbf{x})$, we write $p(\mathbf{x} : \boldsymbol{\tau} \times \boldsymbol{\mu})$ to denote that $\mathbf{x}$ is the collection of variables occurring in the atom, and $p(\boldsymbol{\tau} \times \boldsymbol{\mu})$ is the mode of p.*

By fixing a predicate argument mode to $! \times !$ or to $\star \times !$ we get back to the logic programming input-output behavior, respectively denoted by $+$ and $-$. Several notions of moding have been proposed [1]. We consider here well-moding by extending it to CLP($\mathcal{R}$) programs.

Definition 10 (well-moding). *Let P be a CLP($\mathcal{R}$) program. A clause $p_0(\mathbf{x}_0 : \boldsymbol{\mu}_0 \times \boldsymbol{\tau}_{n+1}) \leftarrow c, p_1(\mathbf{x}_1 : \boldsymbol{\tau}_1 \times \boldsymbol{\mu}_1), \ldots, p_n(\mathbf{x}_n : \boldsymbol{\tau}_n \times \boldsymbol{\mu}_n)$ in P is well-moded if for $i = 1..n+1$, the type assertion $\mathbf{x}_0 : \boldsymbol{\mu}_0, \ldots, \mathbf{x}_{i-1} : \boldsymbol{\mu}_{i-1} \vdash c \rightarrow \mathbf{x}_i : \boldsymbol{\tau}_i$ is valid. P is well-moded if every clause in it is well-moded.*

Example 10. The `MORTGAGE` program is well-moded with moding `mortgage(`$! \times !$`,` $\sqcap \times !$`,` $! \times !$`,` $\star \times !$`)`, which models the first two queries in the introduction. For clause `(m1)` we have to show: $\mathtt{P} :!, \mathtt{T} : \sqcap, \mathtt{R} :! \vdash \mathtt{T = 0,\ B = P} \rightarrow \mathtt{P} :!, \mathtt{T} :!, \mathtt{R} :!, \mathtt{B} :!$ which is immediate. For clause `(m2)`, called c the constraint `T >= 1, NP = P + P * 0.05 - R, NT = T - 1`, we have to show: $\mathtt{P} :!, \mathtt{T} : \sqcap, \mathtt{R} :! \vdash c \rightarrow \mathtt{NP} :!, \mathtt{NT} : \sqcap, \mathtt{R} :!$ and $\mathtt{P} :!, \mathtt{T} : \sqcap, \mathtt{R} :!, \mathtt{NP} :!, \mathtt{NT} :!, \mathtt{B} :! \vdash c \rightarrow \mathtt{P} :!, \mathtt{T} :!, \mathtt{R} :!, \mathtt{B} :!$ which are both readily checked. Analogously, `MORTGAGE` is well-moded with the moding $\mathtt{mortgage}(\star \times \Box,\ \sqcap\times!,\ \Box\times\Box,\ \Box\times\Box)$, which models the third query in the introduction.

We recall that the operational semantics of CLP consists of a transition system from states to states. A state is a pair $\langle Q \| c \rangle$ where Q is a query and c is a constraint, called the constraint store. Initial states are of the form $\langle Q \| \mathtt{true} \rangle$. Final states (if any) are of the form $\langle \varepsilon \| c \rangle$, where ε is the empty query. Well-moding extends to states $\langle Q \| c' \rangle$ by considering the program clause $p \leftarrow c', Q$, where p is a fresh 0-ary predicate.

Definition 11. *A state* $\langle \leftarrow c, p_1(\mathbf{x}_1 : \boldsymbol{\tau}_1 \times \boldsymbol{\mu}_1), \ldots, p_n(\mathbf{x}_n : \boldsymbol{\tau}_n \times \boldsymbol{\mu}_n) \|\ c' \rangle$*, with* $n \geq 0$*, is well-moded if for* $i = 1..n$ *the type assertion* $\mathbf{x}_1 : \boldsymbol{\mu}_1, \ldots, \mathbf{x}_{i-1} : \boldsymbol{\mu}_{i-1} \vdash (c \wedge c') \rightarrow \mathbf{x}_i : \boldsymbol{\tau}_i$ *is valid. A query* Q *is well-moded if the state* $\langle Q \|\ \mathtt{true} \rangle$ *is well-moded.*

Widely studied properties of well-moding in logic programming include persistency along derivations, call pattern characterization and computed answer characterization. They are at the basis of several program analysis, transformation and optimization techniques. The next result shows that the mentioned properties hold for the proposed extension of well-moding to CLP($\mathcal{R}$). By a left-derivation we mean a derivation via the leftmost selection rule.

Theorem 7. *Let* P *be a well-moded CLP($\mathcal{R}$) program and* $Q = \ \leftarrow c, p_1(\mathbf{x}_1 : \boldsymbol{\tau}_1 \times \boldsymbol{\mu}_1), \ldots, p_n(\mathbf{x}_n : \boldsymbol{\tau}_n \times \boldsymbol{\mu}_n)$ *a well-moded query.*

[**persistency**] *Every state selected in a left-derivation of* P *and* Q *is well-moded.*
[**call patterns**] *For every state of the form* $\langle \leftarrow q(\mathbf{x} : \boldsymbol{\tau} \times \boldsymbol{\mu}), R \|\ c' \rangle$ *selected in a left-derivation of* P *and* Q*,* $\vdash c' \rightarrow \mathbf{x} : \boldsymbol{\tau}$ *is valid.*
[**answers**] *For every final state* $\langle \leftarrow \varepsilon \|\ c' \rangle$*,* $\vdash c' \rightarrow \mathbf{x}_1 : \boldsymbol{\mu}_1, \ldots, \mathbf{x}_n : \boldsymbol{\mu}_n$ *is valid.*

Based on these properties, we provide next two examples of the kind of analyses that well-moding allows for.

Example 11. The two queries from the introduction $\leftarrow$ `mortgage(100, 5, 20, B)` and $\leftarrow$ `3 <= T, T <= 5, mortgage(100, T, 20, B)` are well-moded with the moding $\mathtt{mortgage}(!\times!,\ \sqcap\times!,\ !\times!,\ \star\times!)$. By Theorem 7, we conclude definiteness of balance in the answer constraint store. The third query from the introduction $\leftarrow$ `0 <= B, B <= 10, 15 <= R, R <= 20, mortgage(P, 5, R, B)` is well-moded with the moding $\mathtt{mortgage}(\star\times\Box,\ \sqcap\times!,\ \Box\times\Box,\ \Box\times\Box)$. By Theorem 7, we conclude boundedness of principal in the answer constraint store.

Example 12. The full version of the `MORTGAGE` program takes the interest rate as a further predicate argument.

```
(n1)  mortgage(P,T,I,R,B) ←        (n2)  mortgage(P,T,I,R,B) ←
          T = 0,                              T >= 1,
          B = P.                              NP = P + P * I - R,
                                              NT = T - 1,
                                              mortgage(NP,NT,I,R,B).
```

However, this leads to a non-linear constraint appearing in clause (n2). How can we reason on it? We exploit the call pattern characterization property of well-moding by factoring out the `P * I` term.

```
(n2′) mortgage(P,T,I,R,B) ←          (mu)  mult(P,I,M) ←
          T >= 1,                                P * I = M.
          NP = P + M - R,
          NT = T - 1,
          mult(P, I, M),
          mortgage(NP,NT,I,R,B).
```

Consider now as if the predicate `mult` is a built-in of the system, and the input-output properties of Theorem 7 are guaranteed for the mode `mult( !×!, !×!, ⋆×!)`. The rest of the program, namely clauses (n1) and (n2′), is readily checked to be well-moded with moding `mortgage(!×!, ⊓×!, !×!, !×!, ⋆×!)`. Therefore, for every call to `mult` the first and the second arguments are definite, and then the non-linear constraint `P * I = M` becomes linear at run-time.

4 Related Work and Conclusions

A class of formulas, called *parametric queries*, is investigated in [9]. It includes formulas $\exists a \forall \mathbf{v}\ c \rightarrow x \simeq a$, where $\simeq \in \{\leq, =, \geq\}$, or, with our notation, type assertions of the form $\vdash c \rightarrow x : \tau$. The approach switches from the problem of checking $max\{\mathbf{c}^T\mathbf{v} \mid \mathbf{A}_c\mathbf{v} \leq \mathbf{b}_c\} \leq a$ to its dual form $max\{0 \mid \mathbf{y}^T\mathbf{A}_c = \mathbf{c}, a = \mathbf{y}^T\mathbf{b}_c + q, \mathbf{y} \geq \mathbf{0}, q \geq 0\} = 0$, namely on checking feasibility of $\mathbf{y}^T\mathbf{A}_c = \mathbf{c}, a = \mathbf{y}^T\mathbf{b}_c + q, \mathbf{y} \geq \mathbf{0}, q \geq 0$. However, as soon as general type assertions $\mathbf{d}_1 \vdash c \rightarrow \mathbf{d}_2$ are considered, switching to the dual form yields a non-linear problem.

The maximization of a linear function over a parameterized polyhedra is the subject of *(multi)parametric linear programming*. The solution of the problem can be expressed as a piecewise linear function [7] of the parameters. Therefore, an approach alternative to the extraction of the Minkowski's form is to compute (for each variable to be typed) the *max* and *min* functions of a parameteric linear problem and then to compare them on each pair of breaks they are defined on.

Definiteness analysis for CLP($\mathcal{R}$) has been investigated in [2,4,5,8] using abstract interpretation. Compared to well-moding, those approaches *infer* boolean expressions relating definiteness of predicate arguments. E.g., an inferred $x \rightarrow y$ for $p(x, y)$ states that if x is definite when $p(x, y)$ is called then y is definite when it is resolved. However, the mentioned approaches restrict to consider equality constraints only, hence cannot be complete as the type assertion framework.

Summarizing the contribution of the paper, we have introduced a type system for (generalized) linear constraints over the reals that is able to reason about upper bounds, lower bounds and definiteness properties of variables. The problem of checking validity of type assertions has been investigated and solved by proposing two specialized decision procedures. Type assertions are the basic tool for extending well-moding from logic programming to CLP($\mathcal{R}$). We implemented the checking procedure for well-moding, including LPCheck and POLYCheck, in standard C++, relying on the `polylib` library [13] for the calculation of the Minkowski's form of (parameterized) polyhedra (sources and extended technical report at `http://www.di.unipi.it/∼ruggieri/software`). Although a comprehensive assessment over larger programs has to be pursued, our preliminary tests provide us with confidence on the efficiency of the approach in practice.

References

1. Apt, K.R.: From Logic Programming to Prolog. Prentice-Hall, Englewood Cliffs (1997)
2. Baker, N., Søndegaard, H.: Definiteness analysis for CLP($\mathcal{R}$). In: Gupta, G., et al. (eds.) Australian Computer Science Conference, pp. 321–332 (1993)
3. Chernikova, N.V.: An algorithm for finding the general formula for non-negative solutions of systems of linear inequalities. U.S.S.R. Computational Mathematics and Mathematical Physics 5, 228–233 (1965)
4. Codish, M., Genaim, S., Søndegaard, H., Stuckey, P.: Higher-precision groundness analysis. In: Codognet, P. (ed.) ICLP 2001. LNCS, vol. 2237, pp. 135–149. Springer, Heidelberg (2001)
5. de la Banda, M.G., Hermenegildo, M., Bruynooghe, M., Dumortier, V., Janssens, G., Simoens, W.: Global analysis of constraint logic programs. ACM Transactions on Programming Languages and Systems 18(5), 564–614 (1996)
6. Dolzmann, A., Sturm, T., Weispfenning, V.: Real quantifier elimination in practice. In: Matzat, B.H., Greuel, G.-M., Hiss, G. (eds.) Algorithmic Algebra and Number Theory, pp. 221–248. Springer, Berlin (1998)
7. Gal, T.: Postoptimal Analyses, Parametric Programming, and Related Topics, 2nd edn., de Gruyter, Berlin, Germany (1995)
8. Howe, J.M., King, A.: Abstracting numeric constraints with boolean functions. Information Processing Letters 75(1-2), 17–23 (2000)
9. Huynh, T., Joskowicz, L., Lassez, C., Lassez, J.-L.: Practical tools for reasoning about linear constraints. Fundamenta Informaticae 15(3-4), 357–380 (1991)
10. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. Journal of Logic Programming 19, 20, 503–581 (1994)
11. Jaffar, J., Michaylov, S., Stuckey, P., Yap, R.: The CLP($\mathcal{R}$) language and system. ACM Transactions on Programming Languages and Systems 14(3), 339–395 (1992)
12. Lassez, J.-L., McAllon, K.: A canonical form for generalized linear constraints. Journal of Symbolic Computation 13(1), 1–24 (1992)
13. Loechner, V.: Polylib: a library for manipulating parameterized polyhedra, Version 5.22.3 (2007), `http://icps.u-strasbg.fr/polylib/`
14. Loechner, V., Wilde, D.K.: Parameterized polyhedra and their vertices. International Journal of Parallel Programming 25, 525–549 (1997)
15. Murty, K.G.: Linear Programming. John Wiley & Sons, Chichester (1983)
16. Schrijver, A.: Theory of Linear and Integer Programming. J. Wiley & Sons, Chichester (1986)
17. Le Verge, H.: A note on Chernikova's algorithm. Technical Report 635, IRISA, Campus Universitaire de Beaulieu, Rennes, France (1992)

On Polymorphic Recursion, Type Systems, and Abstract Interpretation

Marco Comini[1], Ferruccio Damiani[2], and Samuel Vrech[1]

[1] Dipartimento di Matematica e Informatica, Università di Udine
Via delle Scienze, 206; I-33100 Udine, Italy
[2] Dipartimento di Informatica, Università di Torino
Corso Svizzera, 185; I-10149 Torino, Italy

Abstract. The problem of typing *polymorphic recursion* (i.e., recursive function definitions $\mathsf{rec}\,\{x = e\}$ where different occurrences of x in e are used with different types) has been investigated both by people working on *type systems* and by people working on *abstract interpretation*.

Recently, Gori and Levi have developed a family of abstract interpreters that are able to type all the ML typable recursive definitions and interesting examples of polymorphic recursion. The problem of finding type systems corresponding to their abstract interpreters was open (such systems would lie between the let-free fragments of the ML and of the Milner-Mycroft systems).

In this paper we exploit the notion of principal typing to: (i) provide a complete stratification of (let-free) Milner-Mycroft typability, and (ii) solve the problem of finding type systems corresponding to the type abstract interpreters proposed by Gori and Levi.

Keywords: Principal Typing, Type Inference Algorithm.

1 Introduction

The *Hindley-Milner system* (a.k.a. the *ML type system*) [3], which is the core of the type systems of functional programming languages like SML, OCaml, and Haskell, is only able to infer types for *monomorphic recursion*[1]. The problem of inferring types for *polymorphic recursion*[2] [13,16] has been studied both by people working on type systems [8,12] (see also [1,5,10,11,18]) and by people working on abstract interpretation [6,7,14,15].

Building on results by Cousot [2], Gori and Levi [6,7] have developed a family of type abstract interpreters that are able to type all the ML typable recursive definitions and interesting examples of polymorphic recursion.

As pointed out in [6,7], the problem of finding a type system corresponding to the type abstract interpreter was open. Such type systems would lie between the

[1] I.e., recursive function definitions $\mathsf{rec}\,\{x = e\}$ where all the occurrences of x in e are used with exactly the same type inferred for e.

[2] I.e., recursive function definitions $\mathsf{rec}\,\{x = e\}$ where different occurrences of x in e are used with different types that specialize the type inferred for e.

© Springer-Verlag Berlin Heidelberg 2008

Curry-Hindley system (a.k.a. the *system of simple types*) [9], which is the let-free fragment of the ML type system, and the let-free fragment of the *Milner-Mycroft system* [16]. Previous work of the second author [5] proposes a technique for extending a type system enjoying decidable typability and *principal typings* [10,19] by adding a decidable rule for typing rec-expressions.

In this paper we exploit the technique of [5] to: (1) Develop a family of decidable type systems that lie between the Curry-Hindley type system and (the let-free fragment of) the Milner-Mycroft type system and provide a complete stratification of (let-free) Milner-Mycroft typability; and (2) Solve the problem of finding type systems corresponding to the type abstract interpreters proposed by Gori and Levi [6,7], thus providing a precise characterization of the expressive power of these type abstract interpreters.

Organization of the Paper. Section 2 introduces a core functional programming language (which can be considered the kernel of languages like SML, OCaml, and Haskell) together with other basic definitions that will be used in the rest of the paper. Section 3 develops a family of decidable type systems that provide a complete stratification of (let-free) Milner-Mycroft typability. Section 4 briefly illustrates the Gori-Levi family of type abstract interpreters [6,7] and Section 5 presents a family of type systems that corresponds to these abstract interpreters.

2 Preliminary Definitions

Program *Expressions*, ranged over by e, are defined as $e ::= x \mid c \mid \lambda\, x.e \mid e_1\, e_2 \mid \mathsf{rec}\, \{x = e\}$, where x ranges over program variables (in $\mathbf{P_V}$) and c over constants. *Free* and *bound* occurrences of a variable in an expression are defined as usual. The (finite) set of the free variables of an expression e is denoted by $\mathrm{FV}(e)$.

The set of constants includes the booleans (ranged over by $\flat$), the integer numbers (ranged over by ι), the constructors for pairs (pair) and lists (nil and cons), some logical and arithmetic operators, and the functions for decomposing pairs (fst and snd) and lists (null, hd and tl).

We have omitted conditionals from the syntax of expressions since, for typing purposes, the expression "if e_0 then e_1 else e_2" can be considered as syntactic sugar for the application "ifc e_0 e_1 e_2", where ifc is a constant of suitable type.

The set of *simple types* ($\mathbf{T}_0$), ranged over by u, is defined as $u ::= \mathsf{bool} \mid \mathsf{int} \mid \alpha \mid u_1 \to u_2 \mid u_1 \times u_2 \mid u\, \mathsf{list}$. We have *type variables* (ranged over by α), arrow types, and a selection of *ground types* and *parametric data-types*. The ground types are bool (the type of booleans) and int (the type of integers). The other types are pair types and list types. The constructor $\to$ is right associative and the constructors $\times$ and list bind more tightly than $\to$.

We assume a countable set $\mathbf{T_V}$ of type variables. A *substitution* s is a function from type variables to simple types which is the identity on all but a finite number of type variables. The *domain* of a substitution s is the set of variables $\mathbf{Dom}(\mathsf{s}) \overset{\mathrm{def}}{=} \{\alpha \mid \mathsf{s}(\alpha) \neq \alpha\}$. Substitutions will be denoted by $[\alpha_1 \leftarrow u_1, \ldots, \alpha_n \leftarrow u_n]$ $(n \geq 0)$;

the empty substitution will be denoted by $[]$. The application of a substitution $\mathbf{s}$ to a simple type u, denoted by $\mathbf{s}(u)$, is defined as usual. The *composition* of two substitutions $\mathbf{s}_1$ and $\mathbf{s}_2$ is the substitution, denoted by $\mathbf{s}_1 \cdot \mathbf{s}_2$, such that $\mathbf{s}_1 \cdot \mathbf{s}_2(\alpha) \overset{\text{def}}{=} \mathbf{s}_1(\mathbf{s}_2(\alpha))$, for all type variables α. We say that $\mathbf{s}$ is *more general* than $\mathbf{s}'$, written $\mathbf{s} \leq \mathbf{s}'$, if there is a substitution $\mathbf{s}''$ such that $\mathbf{s}' = \mathbf{s}'' \cdot \mathbf{s}$.

A *type environment* E is a set $\{x_1 : \rho_1, \ldots, x_n : \rho_n\}$ of type assumptions for program variables such that every variable x_i $(1 \leq i \leq n)$ can occur at most once in E. The expression $Dom(E)$ denotes the *domain* of E, which is the set $\{x_1, \ldots, x_n\}$. Given a set of program variables X, the expression $E|_X$ denotes the *restriction of* E *to* X, which is the environment $\{x : \rho \in E \mid x \in X\}$. Given two environments E_1 and E_2, we write $E_1 \oplus E_2$ to denote the environment $E_1 \cup E_2$ under the assumption that $x : \rho_1 \in E_1$ and $x : \rho_2 \in E_2$ imply $\rho_1 = \rho_2$. We write $E, x : \rho$ as short for $E \cup \{x : \rho\}$ under the assumption that $x \notin Dom(E)$. The application of a substitution $\mathbf{s}$ to an environment E, denoted by $\mathbf{s}(E)$, is defined as usual.

Definition 1 (Simple type environments). *A* simple type environment U *is an environment* $\{x_1 : u_1, \ldots, x_n : u_n\}$ *of simple type assumptions for variables.*

According to Wells [19], in a given type system $\vdash$, "a *typing* t for a typable term e is the collection of all the information other than e which appears in the final judgement of a proof derivation showing that e is typable". In this paper we are interested in typings of the shape $\langle U; u\rangle$, where U is a simple type environment and u is a simple type. The following definitions are fairly standard (note that the relation $\leq_{\mathbf{spc}}$ is reflexive and transitive).

Definition 2 (Typing specialization relation $\leq_{\mathbf{spc}}$). *A typing* $\langle U; u\rangle$ *can be specialized to* $\langle U'; u'\rangle$ *(notation* $\langle U; u\rangle \leq_{\mathbf{spc}} \langle U'; u'\rangle$*) if* $\mathbf{s}(U) = U'$ *and* $\mathbf{s}(u) = u'$*, for some substitution* $\mathbf{s}$*. We will write* $\langle U; u\rangle =_{\mathbf{spc}} \langle U'; u'\rangle$ *to mean that both* $\langle U; u\rangle \leq_{\mathbf{spc}} \langle U'; u'\rangle$ *and* $\langle U'; u'\rangle \leq_{\mathbf{spc}} \langle U; u\rangle$ *hold.*

Definition 3 (Principal typings). *Let* $\vdash$ *be a type system with judgements of the shape* $\vdash e : t$*. A typing* t *is* principal for a term e *if* $\vdash e : t$*, and if* $\vdash e : t'$ *implies* $t \leq_{\mathbf{spc}} t'$*. We say that system* $\vdash$ *has the* principal typing property *to mean that every typable term has a principal typing.*

3 A Stratification of (let-free) Milner-Mycroft Typability

In this section we obtain a complete stratification of (let-free) Milner-Mycroft typability by applying the technique of [5] to the Curry-Hindley system [9].

3.1 System $\vdash_0$: Typing the rec-Free Fragment of the Language

The first step of the technique prescribes to take a type system that satisfies the following requirements (parameterized over the actual shape of the typing and of the typing specialization relation).

$$(\textsc{Spc})\ \frac{\vdash e : t}{\vdash e : t'} \text{ where } t \leq_{\mathbf{spc}} t' \qquad (\textsc{Con})\ \vdash c : \langle \emptyset;\, u\rangle \text{ where } u = \mathbf{type}(c) \qquad (\textsc{Var})\ \vdash x : \langle \{x : u\};\, u\rangle \text{ where } u \in \mathbf{T}_0$$

$$(\textsc{App})\ \frac{\vdash e_1 : \langle U_1;\, u_0 \to u\rangle \qquad \vdash e_2 : \langle U_2;\, u_0\rangle}{\vdash e_1\, e_2 : \langle U_1 \oplus U_2;\, u\rangle}$$

$$(\textsc{Abs})\ \frac{\vdash e : \langle U, x : u_0;\, u\rangle}{\vdash \lambda x.e : \langle U;\, u_0 \to u\rangle} \text{ where } x \in \mathrm{FV}(e) \qquad (\textsc{AbsVac})\ \frac{\vdash e : \langle U;\, u\rangle}{\vdash \lambda x.e : \langle U;\, u_0 \to u\rangle} \text{ where } x \notin \mathrm{FV}(e) \text{ and } u_0 \in \mathbf{T}_0$$

Fig. 1. Typing rules for the rec-free fragment of the language (system $\vdash_0$)

c	$\mathbf{type}(c)$	c	$\mathbf{type}(c)$	c	$\mathbf{type}(c)$
$\flat$	bool	not	bool $\to$ bool	fst	$\alpha_1 \times \alpha_2 \to \alpha_1$
ι	int	and, or	bool $\times$ bool $\to$ bool	snd	$\alpha_1 \times \alpha_2 \to \alpha_2$
pair	$\alpha_1 \to \alpha_2 \to (\alpha_1 \times \alpha_2)$	$+, -, *$	int $\times$ int $\to$ int	null	α list $\to$ bool
nil	α list	$=, <$	int $\times$ int $\to$ bool	hd	α list $\to \alpha$
cons	$\alpha \to \alpha$ list $\to \alpha$ list	ifc	bool $\to \alpha \to \alpha \to \alpha$	tl	α list $\to \alpha$ list

Fig. 2. Types for constants

– It has typing judgements of the shape $\vdash e : t$, where the typing t contains assumptions for exactly the variables in $\mathrm{FV}(e)$.
– It has the principal typing property (see Definition 3).
– It is decidable to establish whether a pair t_1 can be specialized to a pair t_2 (i.e., whether $t_1 \leq_{\mathbf{spc}} t_2$ holds).
– There is an algorithm that for every term e decides whether e is typable and, if so, returns a principal typing for e.

System $\vdash_0$ in Fig. 1, which is just a reformulation of the system of simple types [9], satisfies the above requirements.

Rule (Spc), which is the only non-structural rule, allows to specialize (in the sense of Definition 2) the typing inferred for an expression. The rule for typing constants, (Con), uses the function **type** (tabulated in Fig. 2) which specifies a type for each constant. Note that, by rule (Spc), it is possible to assign to a constant c all the specializations of the typing $\langle \emptyset;\, \mathbf{type}(c)\rangle$.

Since $\vdash_0 e : \langle U;\, u\rangle$ implies $Dom(U) = \mathrm{FV}(e)$, we have two rules for typing an abstraction $\lambda x.e$, (Abs) and (AbsVac), corresponding to the two cases $x \in \mathrm{FV}(e)$ and $x \notin \mathrm{FV}(e)$.

3.2 System $\vdash_0^{\mathbf{P}}$: Typing Polymorphic Recursive Definitions

The second step of the technique prescribes to extend system $\vdash_0$ with a typing rule that allows to assign to rec $\{x = e\}$ any typing t that can be assigned to e by assuming the typing t itself for x. This requires to introduce the notion of *typing environment*.

Definition 4 (Typing environments). *A* typing environment D *is an environment* $\{x_1 : t_1, \ldots, x_n : t_n\}$ *of typing assumptions for variables such that*

$$(\textsc{Spc})\ \frac{D \vdash e : t}{D \vdash e : t'} \quad \text{where } t \leq_{\mathbf{spc}} t' \qquad (\textsc{Con})\ D \vdash c : \langle \emptyset;\, u\rangle \quad \text{where } u = \mathbf{type}(c) \qquad (\textsc{Var})\ D \vdash x : \langle \{x : u\};\, u\rangle \quad \text{where } u \in \mathbf{T}_0 \text{ and } x \notin \mathit{Dom}(D)$$

$$(\textsc{Abs})\ \frac{D \vdash e : \langle U, x : u_0;\, u\rangle}{D \vdash \lambda x.e : \langle U;\, u_0 \to u\rangle} \quad \text{where } x \notin D \qquad (\textsc{AbsVac})\ \frac{D \vdash e : \langle U;\, u\rangle}{D \vdash \lambda x.e : \langle U;\, u_0 \to u\rangle} \quad \text{where } x \notin \mathrm{FV}(e),\ u_0 \in \mathbf{T}_0, \text{ and } x \notin D$$

$$(\textsc{App})\ \frac{D \vdash e_1 : \langle U_1;\, u_0 \to u\rangle \qquad D \vdash e_2 : \langle U_2;\, u_0\rangle}{D \vdash e_1\, e_2 : \langle U_1 \oplus U_2;\, u\rangle}$$

$$(\textsc{Rec-P})\ \frac{D, x : \langle U;\, u\rangle \vdash e : \langle U;\, u\rangle}{D \vdash \mathsf{rec}\,\{x = e\} : \langle U;\, u\rangle} \quad \text{where } \mathit{Dom}(U) = \mathrm{FV}_D(\mathsf{rec}\,\{x = e\}) \text{ and } x \notin D \qquad (\textsc{Var-P})\ D, x : t \vdash x : t$$

Fig. 3. Typing rules of system $\vdash_0^{\mathrm{P}}$

$\mathit{Dom}(D) \cap \mathit{VR}(D) = \emptyset$, where $\mathit{VR}(D) \overset{\mathit{def}}{=} \cup_{x:\langle U;\, u\rangle \in D}\, \mathit{Dom}(U)$ is the set of variables occurring in the range of D.

Every typing t occurring in D is implicitly universally quantified over all type variables occurring in t.[3]

The typing rules of system $\vdash_0^{\mathrm{P}}$ (where "P" stands for "polymorphic") are given in Fig. 3. The judgement $D \vdash_0^{\mathrm{P}} e : \langle U;\, u\rangle$ means "e is $\vdash_0^{\mathrm{P}}$-typable in D with typing $\langle U;\, u\rangle$", where D is a typing environment specifying typing assumptions for variables that may or may not occur free in e, and

$\langle U;\, u\rangle$ is the typing inferred for e, where U is a simple type environment containing the type assumptions for the free variables of e which are not in $\mathit{Dom}(D)$, and u is a simple type.

Let D be a typing environment, "$x \notin D$" is short for "$x \notin \mathit{Dom}(D) \cup \mathit{VR}(D)$" and $\mathrm{FV}_D(e) \overset{\mathrm{def}}{=} (\mathrm{FV}(e) - \mathit{Dom}(D)) \cup \mathit{VR}(D|_{\mathrm{FV}(e)})$ is the *set of the free variables of the expression e in D*. In any valid judgement $D \vdash_0^{\mathrm{P}} e : \langle U;\, u\rangle$ it holds that $\mathit{Dom}(D) \cap \mathit{Dom}(U) = \emptyset$ and $\mathit{Dom}(U) = \mathrm{FV}_D(e)$.

Rules (Spc), (Con), (Var), (Abs), (AbsVac), and (App) are just the rules of system $\vdash_0$ (in Fig. 1) modified by adding the typing environment D on the left of the typing judgements and, when necessary, side conditions (like "$x \notin \mathit{Dom}(D)$" in rule (Var)) to ensure that $\mathit{Dom}(D) \cap \mathit{Dom}(U) = \emptyset$.

Rule (Rec-P) allows to assign to a recursive definition $\mathsf{rec}\,\{x = e\}$ any typing t that can be assigned to e by assuming the typing t for x. Note that the combined use of rules (Var-P) and (Spc) allows to assign different specializations $t_i = \langle U_i;\, u_i\rangle$ $(1 \leq i \leq n)$ of $t = \langle U;\, u\rangle$ to different occurrences of x in e, provided that $\oplus_{1 \leq i \leq n} U_i$ is defined.

The following theorem (taken from [5]) shows that system $\vdash_0^{\mathrm{P}}$ has the same expressive power as the Milner-Mycroft system [16]. This implies that typability in system $\vdash_0^{\mathrm{P}}$ is undecidable (as is in the Milner-Mycroft system [8,12]).

[3] To emphasize this fact we might have used assumptions of the shape $\forall\overrightarrow{\alpha}.t$ where $\overrightarrow{\alpha}$ is the sequence of all the type variables occurring in t.

Theorem 5 ([5]). *Let e be a closed expression. Then $\emptyset \vdash_0^{\mathrm{P}} e : \langle\emptyset;\, u\rangle$ if and only if $\emptyset \vdash e : u$ is Milner-Mycroft derivable.*

3.3 Systems $\vdash_0^k$ ($k \geq 1$): A Family of Decidable Restrictions of $\vdash_0^{\mathrm{P}}$

The third step of the technique prescribes to introduce a family of decidable restrictions of rule (Rec-P). This requires to introduce the notion of *principal-in-D typing*, which adapts the notion of principal typing (see Definition 3) to deal with the typing environment D.

Definition 6 (Principal-in-D typings). *Let $\vdash$ be a system with judgements of the shape $D \vdash e : t$. A typing t is* principal-in-D *for a term e if $D \vdash e : t$, and if $D \vdash e : t'$ implies $t \leq_{\mathbf{spc}} t'$. We say that system $\vdash$ has the* principal-in-D typing property *to mean that every typable term has a principal-in-D typing.*

For every finite set of variables $X = \{x_1, \ldots, x_n\}$ ($n \geq 0$) let $\mathsf{T}_X \overset{\text{def}}{=} \{\langle U;\, u\rangle \mid \langle U;\, u\rangle$ is typing such that $\mathit{Dom}(U) = X\}$, and $\mathsf{B}_X \overset{\text{def}}{=} \{\langle\{x_1 : \alpha_1, \ldots, x_n : \alpha_n\};\, \alpha\rangle \mid$ the type variables $\alpha_1, \ldots, \alpha_n, \alpha$ are all distinct$\} \subseteq \mathsf{T}_X$. The typing specialization relation $\leq_{\mathbf{spc}}$ (see Definition 2) is a preorder over T_X and, for all typings $b \in \mathsf{B}_X$ and $t \in \mathsf{T}_X$, $b \leq_{\mathbf{spc}} t$. Moreover for every subset S_X of T_X,

$$\mathbf{Min}_{\leq_{\mathbf{spc}}}(\mathsf{S}_X) \overset{\text{def}}{=} \{t \in \mathsf{S}_X \mid t \leq_{\mathbf{spc}} t' \text{ for all } t' \in \mathsf{S}_X\}$$

is the (possibly empty) *set of the $\leq_{\mathbf{spc}}$-minimum elements of* S_X.

The following proposition holds.

Proposition 7. *Let $\vdash$ be a system with judgements of the shape $D \vdash e : t$. A typing t is a principal-in-D typing for a term e if and only if $t \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t' \mid D \vdash e : t'\})$.*

For every $k \geq 1$, let $\vdash_0^k$ be the system obtained from $\vdash_0^{\mathrm{P}}$ by replacing rule (Rec-P) with the rule:

$$(\textsc{Rec-}k)\ \frac{D, x : t_0 \vdash e : t_1 \ \cdots\ D, x : t_{k-1} \vdash e : t_k}{D \vdash \mathsf{rec}\,\{x = e\} : t_k} \text{ where}$$

$$\begin{array}{l} x \notin D,\ t_0 \in \mathsf{B}_{\mathrm{FV}_D(\mathsf{rec}\,\{x=e\})}, \\ (\forall i \in \{1, \ldots, k\})\, t_i \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t \mid D, x : t_{i-1} \vdash e : t\}),\ t_{k-1} = t_k \end{array} \tag{3.1}$$

(note that $D \vdash_0^k e : t$ implies $D \vdash_0^{k+1} e : t$). According to Proposition 7, in rule (Rec-k), the requirement (3.1) is equivalent to *(for all $i \in \{1, \ldots, k\}$) t_i is a principal-in-$(D, x : t_{i-1})$ typing for e*. Therefore, as pointed out in [5], the checking of a purported $\vdash_0^k$ derivation requires the ability to decide whether a typing is principal-in-D. Note that requirement (3.1) is crucial. In fact, removing it would make rule (Rec-k) equivalent to rule (Rec-P) for all $k \geq 2$.

For all $k \geq 1$, system $\vdash_0^k$ has the principal-in-D typing property and $\vdash_0^k$-typability is decidable — see the explanations before Theorem 10.

An Inference Algorithm for $\vdash_0^k$ ($k \geq 1$). A *unification problem* P is a set of equalities between simple types. A *solution* to P (*unifier*) is an idempotent substitution $\mathbf{s}$ such that $\mathbf{s}(u_1) = \mathbf{s}(u_2)$ for all $(u_1 = u_2) \in P$. A *Most General Unifier* is a solution minimal w.r.t. $\leq$ (and thus all Most General Unifiers are equivalent up to renaming of type variables). We will write $\mathbf{MGU}(P)$ for the set of all the most general unifiers for P and $\mathbf{mgu}(P)$ for any element of $\mathbf{MGU}(P)$.

The inference algorithm makes use of an algorithm for checking whether the $\leq_{\mathbf{spc}}$ relation (see Definition 2) holds and of the standard algorithm for finding a most general solution to a unification problem. Note that the first algorithm is a particular case of the latter.

The inference algorithm is presented by defining (for all $k \geq 1$) a function $\mathbf{PT}_0^k$ which, for every expression e and environment D, returns a set of typings $\mathbf{PT}_0^k(D, e)$ such that $\mathbf{PT}_0^k(D, e) = \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t \mid D \vdash_0^k e : t\})$.

Definition 8 (Inductive characterization of the set of principal-in-D typings for e w.r.t. $\vdash_0^k$). *For every expression e and environment D, the set $\mathbf{PT}_0^k(D, e)$ is defined by structural induction on e.*

$\underline{e = x}\rfloor$ *If $x : \langle U; u\rangle \in D$ and the substitution $\mathbf{s}$ is a renaming of $\overrightarrow{\alpha} = \mathrm{FTV}(U) \cup \mathrm{FTV}(u)$ with fresh type variables, then $\langle \mathbf{s}(U); \mathbf{s}(u)\rangle \in \mathbf{PT}_0^k(D, x)$.*
If $x \notin Dom(D)$ and α is a type variable, then $\langle \{x : \alpha\}; \alpha\rangle \in \mathbf{PT}_0^k(D, x)$.

$\underline{e = c}\rfloor$ *If $\mathbf{type}(c) = u$, then $\langle \emptyset; u\rangle \in \mathbf{PT}_0^k(D, c)$.*

$\underline{e = \lambda x.e_0}\rfloor$ *If $\langle U; u_0\rangle \in \mathbf{PT}_0^k(D, e_0)$, then*
- *If $x \notin \mathrm{FV}(e_0)$ and α is a fresh type variable, then $\langle U; \alpha \to u_0\rangle \in \mathbf{PT}_0^k(D, \lambda x.e_0)$.*
- *If $x \in \mathrm{FV}(e_0)$ and $U = U', x : u$, then $\langle U'; u \to u_0\rangle \in \mathbf{PT}_0^k(D, \lambda x.e_0)$.*

$\underline{e = e_0 e_1}\rfloor$ *If $\langle U_0; u_0\rangle \in \mathbf{PT}_0^k(D, e_0)$, then*
- *If $u_0 = \alpha$ (a type variable), α_1 and α_2 are fresh type variables, $\langle U_1; u_1\rangle \in \mathbf{PT}_0^k(D, e_1)$ is renamed apart, and $\mathbf{s} \in \mathbf{MGU}(\{u_1 = \alpha_1, \alpha = \alpha_1 \to \alpha_2\} \cup \{u' = u'' \mid x : u' \in U_0 \text{ and } x : u'' \in U_1\}$, then $\langle \mathbf{s}(U_0) \oplus \mathbf{s}(U_1); \mathbf{s}(\alpha_2)\rangle \in \mathbf{PT}_0^k(D, e_0 e_1)$.*
- *If $u_0 = u_2 \to u$, the pair $\langle U_1; u_1\rangle \in \mathbf{PT}_0^k(D, e_1)$ is renamed apart, and $\mathbf{s} \in \mathbf{MGU}(\{u_1 = u_2\} \cup \{u' = u'' \mid x : u' \in U_0 \text{ and } x : u'' \in U_1\})$, then $\langle \mathbf{s}(U_0) \oplus \mathbf{s}(U_1); \mathbf{s}(u)\rangle \in \mathbf{PT}_0^k(D, e_0 e_1)$.*

$\underline{e = \mathsf{rec}\,\{x = e_0\}}\rfloor$ *If $h \in \{1, \ldots, k\}$, $t_0 \in \mathsf{B}_{\mathrm{FV}_D(e)}$, $t_1 \in \mathbf{PT}_0^k((D, x : t_0), e_0)$, $\ldots$, $t_h \in \mathbf{PT}_0^k((D, x : t_{h-1}), e_0)$, $t_1 \not\leq_{\mathbf{spc}} t_0$, $\ldots$ $t_{h-1} \not\leq_{\mathbf{spc}} t_{h-2}$, and $t_h \leq_{\mathbf{spc}} t_{h-1}$, then $t_{h-1} \in \mathbf{PT}_0^k(D, e)$.*

For every $k \geq 1$, expression e, and typing environment D, the set $\mathbf{PT}_0^k(D, e)$ is an equivalence class of typings modulo renaming of the type variables in a typing. The following proposition holds.

Proposition 9. *For every $k \geq 1$, expression e and environment D, if $\langle U; u\rangle \in \mathbf{PT}_0^k(D, e)$, then*

1. *$Dom(U) = \mathrm{FV}_D(e)$, and*
2. *$\langle U'; u'\rangle \in \mathbf{PT}_0^k(D, e)$ if and only if there is a bijection $\mathbf{s} : \mathbf{T_V} \to \mathbf{T_V}$ such that $\mathbf{s}(U) = U'$ and $\mathbf{s}(u) = u'$.*

Indeed Definition 8 specifies a sound, complete, and terminating inference algorithm: to perform type inference on e w.r.t. D simply follow the definition of $\mathbf{PT}_0^k(D, e)$, choosing fresh type variables and using the unification and $\leq_{\mathbf{spc}}$-checking algorithms as necessary.

Theorem 10 (Soundness and completeness of $\mathbf{PT}_0^k$ for $\vdash_0^k$). *For every $k \geq 1$, expression e, and environment D:*
If $t \in \mathbf{PT}_0^k(D, e)$, then $D \vdash_0^k e : t$.
If $D \vdash_0^k e : t'$, then $t \leq_{\mathbf{spc}} t'$ for some $t \in \mathbf{PT}_0^k(D, e)$.

Corollary 11. *For every $k \geq 1$, expression e, and environment D:*
$\mathbf{PT}_0^k(D, e) = \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t \mid D \vdash_0^k e : t\})$.

Comparison with System $\vdash_{\mathbf{0}}^{\mathbf{P}}$. The relation between system $\vdash_0^k$ and system $\vdash_0^{\mathrm{P}}$ is stated by the following theorems. Roughly speaking, the first says that when rule (Rec-k) works at all, it works as well as rule (Rec-P) does, and the second says that the family of systems $\vdash_0^k$ ($k \geq 1$) provides a complete stratification of $\vdash_0^{\mathrm{P}}$-typability.

Theorem 12. *For every $k \geq 1$:*
If $D \vdash_0^k e : t$, then $D \vdash_0^{\mathrm{P}} e : t$.
If e is $\vdash_0^k$-typable in D and $D \vdash_0^{\mathrm{P}} e : t$, then $D \vdash_0^k e : t$.

Theorem 13. *If $D \vdash_0^{\mathrm{P}} e : t$, then there exists $k \geq 1$ such that $D \vdash_0^k e : t$.*

Note that Theorem 10 (which implies that, for all $k \geq 1$, system $\vdash_0^k$ has the principal-in-D typing property), Theorem 13, and Theorem 12.2, imply that system $\vdash_0^{\mathrm{P}}$ has the principal-in-D typing property.

Comparison with the ML Type System. For all $k \geq 1$, system $\vdash_0^k$ is able to type recursive definitions that are not ML-typable. The examples for $k = 1$ are not particularly interesting: the prototypical term is the always divergent function rec $\{x = x\,x\}$, that has principal-in-$\emptyset$ typing $\langle\emptyset; \alpha\rangle$). Instead, with $k = 2$ it is already possible to type many interesting examples of polymorphic recursion. Consider for instance the OCaml program (taken from [17])

```
type 'a seq = EMPTY | SEQ of 'a * ('a * 'a) seq ;;

let rec size s = match s with
                   EMPTY     -> 0
                 | SEQ(x,ps) -> 1 + 2 * (size ps) ;;
```

where `'a seq` is a polymorphic sequence type (a sequence is either empty or made of an element paired with a sequence of pairs of elements) and `size` is a function that returns the number of elements contained in a sequence. Although OCaml allows the definition of the `'a seq` recursive data-type, the ML type system (and, therefore, OCaml) is not able to type the function `size`. Instead, for all $k \geq 2$, rule (Rec-k) is able to assign the principal-in-$\emptyset$ pair `<{},'a seq -> int>` to the function `size`.

The following example shows that, for all $k \geq 1$, system $\vdash_0^k$ is not able to type all the ML-typable recursive definitions.

Example 14. The ML-typable term $\mathsf{rec}\,\{f = \lambda\, g\, y.\mathsf{if\,false\,then}\, y\, \mathsf{else}\, g\,(f\, g\, y)\}$ is not $\vdash_0^2$-typable but $\vdash_0^3$-typable with principal-in-$\emptyset$ typing $\langle\emptyset;\, (\alpha \to \alpha) \to \alpha \to \alpha\rangle$. The ML-typable term $\mathsf{rec}\,\{f = \lambda\, g\, h_1\, y.\mathsf{if\,false\,then}\, y\, \mathsf{else}\, g\,(f\, g\, h_1\,(f\, h_1\, g\, y))\}$ is not $\vdash_0^3$-typable but $\vdash_0^4$-typable with principal-in-$\emptyset$ typing $\langle\emptyset;\, (\alpha \to \alpha) \to (\alpha \to \alpha) \to \alpha \to \alpha\rangle$. In general, for all $m \geq 0$, the ML-typable term

$$\begin{aligned}&\mathsf{rec}\,\{f = \lambda\, g\, h_1\, h_2 \cdots h_m\, y.\,\mathsf{if\,false\,then}\, y\, \mathsf{else}\\ &\qquad\qquad g\,(f\, g\, h_1\, h_2 \cdots h_m\,(f\, h_1\, g\, h_2 \cdots h_m\,(\cdots(f\, h_1 \cdots h_{m-1}\, g\, y)\cdots))\}\end{aligned}$$

is not $\vdash_0^{m+2}$-typable but $\vdash_0^{m+3}$-typable with principal-in-$\emptyset$ typing

$$\langle\emptyset;\, \underbrace{(\alpha \to \alpha) \to (\alpha \to \alpha) \to \cdots \to (\alpha \to \alpha)}_{m+1} \to \alpha \to \alpha\rangle.$$

3.4 Systems $\vdash_0^{k,\mathrm{ML}}$ ($k \geq 1$): Recovering ML-Typability

The fourth step of the technique allows to extend system $\vdash_0^k$ (for any given $k \geq 1$) to type not $\vdash_0^k$-typable expressions that can be typed by a given decidable restriction of $\vdash_0^{\mathrm{P}}$, while preserving decidable typability and principal-in-D typings.

We say that *a typing rule for recursive definitions* (Rec-?) *is* $\vdash_0^{\mathrm{P}}$*-suitable* to mean that the system $\vdash_0^?$ obtained from $\vdash_0^{\mathrm{P}}$ by replacing rule (Rec-P) with rule (Rec-?):

1. is a restriction of system $\vdash_0^{\mathrm{P}}$ (i.e., $D \vdash_0^? e : \langle U;\, u\rangle$ implies $D \vdash_0^{\mathrm{P}} e : \langle U;\, u\rangle$),
2. has the principal-in-D typing property, and
3. there is an algorithm that, given a typing environment D and a term e, returns a principal-in-D typing for e.

Theorems 10 and 12 guarantee that, for all $k \geq 1$, adding to system $\vdash_0^k$ a $\vdash_0^{\mathrm{P}}$-suitable rule (Rec-?) with an additional side condition ensuring that the rule can be applied only if rule (Rec-k) is not applicable, results in a system, denoted by $\vdash_0^{k,?}$, with both decidable typability and principal-in-D typing property.

So, to extend system $\vdash_0^k$ to type all the ML typable recursive definitions, we have just to add to system $\vdash_0^k$ a $\vdash_0^{\mathrm{P}}$-suitable rule which (without the additional side condition) is at least as expressive as the rule for monomorphic recursions. The simplest way of doing this would be to add (a version, modified to fit into system $\vdash_0^k$, of) the ML rule itself:

$$(\textsc{Rec-ML})\ \frac{D \vdash e : \langle U, x : u;\, u\rangle}{D \vdash \mathsf{rec}\,\{x = e\} : \langle U;\, u\rangle}\quad \text{where } x \notin D$$

and to restrict it with the additional side condition:

$$\boxed{\begin{array}{l}\text{and there are no } t_0, t_1, \cdots, t_k \text{ such that:}\\ \quad t_0 \in \mathsf{B}_{\mathrm{FV}_D(\mathsf{rec}\,\{x=e\})},\\ \quad (\text{for all } i \in \{1, \ldots, k\})\ t_i \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t \mid D, x : t_{i-1} \vdash e : t\}), \text{ and } t_{k-1} = t_k\end{array}}$$

(2) $H \rhd x \Rightarrow H(x)$
where $x \in \mathbf{P_V}$

$$(5)\ \frac{H \rhd e_1 \Rightarrow (u_1, \mathbf{s}_1) \qquad H \rhd e_2 \Rightarrow (u_2, \mathbf{s}_2) \\ \mathbf{s} = \mathbf{mgu}\left(\{u_1 = f_1 \to f_2, u_2 = f_1\} \cup \mathbf{eqs}(\mathbf{s}_1) \cup \mathbf{eqs}(\mathbf{s}_2)\right)}{H \rhd e_1 e_2 \Rightarrow (\mathbf{s}(f_2), \mathbf{s})}$$

$$(6)\ \frac{H[x \leftarrow (f_1, \epsilon)] \rhd e \Rightarrow (u, \mathbf{s}) \qquad u_1 = \mathbf{s}(f_1)}{H \rhd (\lambda x.e) \Rightarrow ((u_1 \to u), \mathbf{s})}$$

$$(7)\ \frac{H \rhd \mathsf{rec}\{x = e\} \Rightarrow^n_{T_P} (u_n, \mathbf{s}_n) \\ H \rhd \mathsf{rec}\{x = e\} \Rightarrow^{n+1}_{T_P} (u_{n+1}, \mathbf{s}_{n+1}) \\ (u_n, \mathbf{s}_n) = (u_{n+1}, \mathbf{s}_{n+1})}{H \rhd \mathsf{rec}\{x = e\} \Rightarrow (u_n, \mathbf{s}_n)}$$

(8) $H \rhd \mathsf{rec}\{x = e\} \Rightarrow^0_{T_P} (\alpha, \epsilon)$
with $\alpha \in \mathbf{T_V}$ fresh

$$(9)\ \frac{H \rhd \mathsf{rec}\{x = e\} \Rightarrow^{n-1}_{T_P} (u_1, \mathbf{s}_1) \\ H \rhd \mathsf{rec}\{x = e\}(u_1, \mathbf{s}_1) \Rightarrow_{T_P} (u_2, \mathbf{s}_2)}{H \rhd \mathsf{rec}\{x = e\} \Rightarrow^n_{T_P} (u_2, \mathbf{s}_2)}$$

$$(10)\ \frac{H[x \leftarrow (u, \mathbf{s})] \rhd e \Rightarrow (u_1, \mathbf{s}_1)}{H \rhd \mathsf{rec}\{x = e\}(u, \mathbf{s}) \Rightarrow_{T_P} (u_1, \mathbf{s}_1)}$$

(1) $H \rhd c \Rightarrow (\mathit{type}(c), \epsilon)$

Fig. 4. Rules of the Gori-Levi type abstract interpreter $\rhd_{\mathrm{GL}}$

which ensures that the rule can be applied only when rule (Rec-k) is not applicable. Let $\vdash_0^{k,\mathrm{ML}}$ denote the resulting system.

A sound and complete inference algorithm for system $\vdash_0^{k,\mathrm{ML}}$ ($\mathbf{PT}_0^{k,\mathrm{ML}}$) can be obtained from the inference algorithm $\mathbf{PT}_0^k$, given in Definition 8, by adding the following sub-clause to the clause for rec-expression.

> Otherwise, if $\langle U, x : u;\ u' \rangle \in \mathbf{PT}_0^k(D, e_0)$, and $\mathbf{s} \in \mathbf{MGU}(\{u = u'\})$, then $\mathbf{s}(\langle U;\ u \rangle) \in \mathbf{PT}_0^k(D, e)$.

Theorem 15 (Soundness and completeness of $\mathbf{PT}_0^{k,\mathbf{ML}}$ w.r.t. $\vdash_0^{k,\mathbf{ML}}$). *For every $k \geq 1$, expression e, and environment D:*
(Soundness) *If $t \in \mathbf{PT}_0^{k,\mathrm{ML}}(D, e)$, then $D \vdash_0^{k,\mathrm{ML}} e : t$.*
(Completeness) *If $D \vdash_0^{k,\mathrm{ML}} e : t'$, then $t \leq_{\mathbf{spc}} t'$ for some $t \in \mathbf{PT}_0^{k,\mathrm{ML}}(D, e)$.*

Corollary 16. *For every $k \geq 1$, expression e, and environment D:*
$\mathbf{PT}_0^k(D, e) = \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t \mid D \vdash_0^{k,\mathrm{ML}} e : t\})$.

4 The Gori-Levi Type Abstract Interpreter Revisited

In this section we briefly recall the type abstract interpreter of Gori and Levi [6,7]. The syntax of the small ML-like language used in the present paper is slightly different from the one in [6,7]. Namely, our language uses the rec-notation instead of the μ-notation, uses booleans instead of integers in the test of conditionals, and has more constants. In order to simplify the presentation of the correspondence result (see Section 5.2) we reformulate in the following the type abstract interpreter by using the language syntax and the notations we used so far.

The *abstract semantics* (or *abstract typing*) of an expression e is the type $u \in \mathbf{T}_0$ of the expression e together with a substitution $\mathbf{s} \in \mathbf{T_V} \to \mathbf{T}_0$ representing a constraint on the type variables. It is computed w.r.t. a given *abstract*

$$(11)\ \frac{H \triangleright \mathsf{rec}\{x=e\} \Rightarrow^{k}_{\text{wid}} (u,\mathbf{s})}{H \triangleright \mathsf{rec}\{x=e\} \Rightarrow (u,\mathbf{s})} \qquad (12)\ \frac{\begin{array}{c} H \triangleright \mathsf{rec}\{x=e\} \Rightarrow^{k-1}_{T_P} (u_1,\mathbf{s}_1) \\ H \triangleright \mathsf{rec}\{x=e\}(u_1,\mathbf{s}_1) \Rightarrow_{T_P} (u_2,\mathbf{s}_2) \\ (u_1,\mathbf{s}_1)=(u_2,\mathbf{s}_2) \end{array}}{H \triangleright \mathsf{rec}\{x=e\} \Rightarrow^{k}_{\text{wid}} (u_1,\mathbf{s}_1)}$$

$$(13)\ \frac{\begin{array}{cc} H \triangleright \mathsf{rec}\{x=e\} \Rightarrow^{k-1}_{T_P} (u_1,\mathbf{s}_1) & H \triangleright \mathsf{rec}\{x=e\}(u_1,\mathbf{s}_1) \Rightarrow_{T_P} (u_2,\mathbf{s}_2) \\ (u_1,\mathbf{s}_1) \neq (u_2,\mathbf{s}_2) & \mathbf{s} = \mathbf{mgu}(\{\mathbf{s}_2(u_1)=u_2\} \cup \mathbf{eqs}(\mathbf{s}_2)) \end{array}}{H \triangleright \mathsf{rec}\{x=e\} \Rightarrow^{k}_{\text{wid}} (\mathbf{s}(u_1),\mathbf{s})}$$

Fig. 5. Rules of the Gori-Levi decidable type abstract interpreter $\triangleright^{k,\text{ML}}_{\text{GL}}$

environment H, which is a partial function mapping program variables $\mathbf{P_V}$ to abstract typings $(\mathbf{T}_0 \times (\mathbf{T_V} \to \mathbf{T}_0))$. The intuition is that the substitution $\mathbf{s} \in \mathbf{T_V} \to \mathbf{T}_0$ collects all the constraints generated on the type variables of H while inferring the type u for the expression e.

Gori and Levi obtained systematically the rules of a non-effective type abstract interpreter $\triangleright_{\text{GL}}$ by abstracting the concrete semantics on the type domain $\mathbf{P_V} \to (\mathbf{T}_0 \times (\mathbf{T_V} \to \mathbf{T}_0))$. These rules are (modulo the change of notation) those in Fig. 4. Rule numbers correspond exactly to those in [6,7]. Note that, rule (3) (for integer additions) and rule (4) (for conditionals) are missing because they are encompassed by the use of the function **type** for typing constants in rule (1) — see the discussion in Section 3.1. The operation **eqs** (used in rule (5)) is defined as $\mathbf{eqs}([\alpha_1 \leftarrow u_1, \ldots, \alpha_n \leftarrow u_n]) \overset{\text{def}}{=} \{\alpha_1 = u_1, \ldots, \alpha_n = u_n\}$, the operation **mgu** returns a most general unifier, and $H[x \leftarrow \rho]$ (used in rules (6) and (10)) is a destructive update.

As this interpreter is possibly non-effective, because the type domain is not Nötherian, Gori and Levi [6,7] replace rule (7) with the rules (11), (12), (13) of Fig. 5 (that arises from a family of widening operators as k varies) yielding the effective type abstract interpreter $\triangleright^{k,\text{ML}}_{\text{GL}}$.

5 The Type Systems Corresponding to $\triangleright_{\text{GL}}$ and $\triangleright^{k,\text{ML}}_{\text{GL}}$

In this section we solve the open problem [6,7] of finding type systems corresponding to the $\triangleright_{\text{GL}}$ and $\triangleright^{k,\text{ML}}_{\text{GL}}$ type abstract interpreters.

As stated in [6,7], the type systems corresponding to the abstract interpreters $\triangleright_{\text{GL}}$ and $\triangleright^{k,\text{ML}}_{\text{GL}}$ would lie between the Curry-Hindley and Milner-Mycroft type systems. In Section 5.1 we will define (less powerful) variants of systems $\vdash^{k,\text{ML}}_{0}/\vdash^{\text{P}}_{0}$, that we will call $\vdash^{k,\text{ML}}_{\text{GL}}/\vdash_{\text{GL}}$, by requiring that all the occurrences of x in e are used with *the same* type. In Section 5.2 we will prove that systems $\vdash^{k,\text{ML}}_{\text{GL}}/\vdash_{\text{GL}}$ are equivalent to the $\triangleright^{k,\text{ML}}_{\text{GL}}/\triangleright_{\text{GL}}$ abstract interpreters. I.e., we will prove that the type abstract interpreter $\triangleright_{\text{GL}}$ infers a type to a recursive function definition $\mathsf{rec}\{x = e\}$ by requiring that all the occurrences of x in e are used with a same type (as in the Curry-Hindley type system) that specializes (as in the Milner-Mycroft type system) the type inferred for e. This result disproves the misconception that in the type system corresponding to the abstract interpreters $\triangleright^{k,\text{ML}}_{\text{GL}}/\triangleright_{\text{GL}}$ "the different function applications of a recursive function can lead to different (but *compatible*) instantiation of the recursive function type" ([7], page 142). Namely, our result shows that two different instantiations are *compatible* if and only if they are *the same*. Finally, in Section 5.3, we provide a simpler type system that has the same expressive power of the non-effective abstract interpreter $\rhd_{\mathrm{GL}}$.

Example 17. The non ML-typable term (taken from [2] — see also [7], page 140)

$$\begin{array}{rcll} \mathsf{rec}\,\{\, f & = & \lambda\, f_1\, g\, n\, x.\mathsf{if}\ n=0\ \mathsf{then} & g\,x \\ & & \phantom{\lambda\, f_1\, g\, n\, x.\mathsf{if}\ n=0\ }\mathsf{else} & f\ \ f_1\ \ (\lambda\, y.(\lambda\, h.g\,(h\,y)))\ \ (n-1)\ \ x\ \ f_1\ \} \end{array}$$

(which defines the function F such that $F\,f_1\,g\,n\,x = g\,\overbrace{(f_1\cdots(f_1}^{n\ \text{times}}\ x)\cdots))$ is typed by $\rhd_{\mathrm{GL}}$ (and, for $k\geq 3$, also by $\rhd_{\mathrm{GL}}^{k,\mathrm{ML}}$) with type $u=(\alpha\rightarrow\alpha)\rightarrow(\alpha\rightarrow\beta)\rightarrow\mathsf{int}\rightarrow\alpha\rightarrow\beta$. The function f is recursively called with the specialized type $(\alpha\rightarrow\alpha)\rightarrow(\alpha\rightarrow(\alpha\rightarrow\alpha)\rightarrow\beta)\rightarrow\mathsf{int}\rightarrow\alpha\rightarrow(\alpha\rightarrow\alpha)\rightarrow\beta$ (obtained from u by substituting β with $(\alpha\rightarrow\alpha)\rightarrow\beta$). The typing $\langle\emptyset;\,u\rangle$ is principal-in-$\emptyset$ w.r.t. $\vdash_0^{\mathrm{P}}$ (and, for $k\geq 3$, also w.r.t. $\vdash_0^{k,\mathrm{ML}}$).

The non ML-typable term $\mathsf{rec}\,\{g=\lambda\,x\,y.(g\,(\lambda\,z.x)\,(\lambda\,z.y))+(g\,(\lambda\,z.y)\,(\lambda\,z.x))\}$ is $\vdash_0^{\mathrm{P}}$-typable and, for $k\geq 2$, also $\vdash_0^{k,\mathrm{ML}}$-typable with principal-in-$\emptyset$ typing $\langle\emptyset;\,\alpha\rightarrow\beta\rightarrow\mathsf{int}\rangle$ (the function g is recursively called with the typings $\langle\emptyset;\,(\alpha'\rightarrow\alpha)\rightarrow(\beta'\rightarrow\beta)\rightarrow\mathsf{int}\rangle$ and $\langle\emptyset;\,(\beta'\rightarrow\beta)\rightarrow(\alpha'\rightarrow\alpha)\rightarrow\mathsf{int}\rangle$). However, the $\rhd_{\mathrm{GL}}$ and, for $k\geq 2$, the $\rhd_{\mathrm{GL}}^{k,\mathrm{ML}}$ type abstract interpreters assign to the term the type $\alpha\rightarrow\alpha\rightarrow\mathsf{int}$.

5.1 Systems $\vdash_{\mathbf{GL}}$, $\vdash_{\mathbf{GL}}^{\boldsymbol{k}}$, and $\vdash_{\mathbf{GL}}^{\boldsymbol{k},\mathbf{ML}}$

Given a typing environment D and an expression e, the set $\mathrm{FV}_D^{\mathrm{GL}}(e)\stackrel{\mathrm{def}}{=}\mathrm{FV}(e)\cup \mathit{VR}(D|_{\mathrm{FV}(e)})$ is the *set of the free variables of e and of the free variables of e in D* (note that $\mathrm{FV}_D^{\mathrm{GL}}(e)=\mathrm{FV}(e)\cup\mathrm{FV}_D(e)$). The rules of system $\vdash_{\mathrm{GL}}$ are obtained by those of system $\vdash_0^{\mathrm{P}}$ (in Fig. 3) by replacing the rules (Var-P) and (Rec-P) with the following three rules:

(Var-GL) $D,\,x:\langle U;\,u\rangle\vdash x:\langle U, x:u;\,u\rangle$

(Rec-GL) $\dfrac{D,x:\langle U;\,u\rangle\vdash e:\langle U,x:u';\,u\rangle}{D\vdash\mathsf{rec}\,\{x=e\}:\langle U;\,u\rangle}$
where $\mathit{Dom}(U)=\mathrm{FV}_D^{\mathrm{GL}}(\mathsf{rec}\,\{x=e\})$
and $x\notin D$

(RecVac-GL) $\dfrac{D,x:\langle U;\,u\rangle\vdash e:\langle U;\,u\rangle}{D\vdash\mathsf{rec}\,\{x=e\}:\langle U;\,u\rangle}$
where $\mathit{Dom}(U)=\mathrm{FV}_D^{\mathrm{GL}}(\mathsf{rec}\,\{x=e\})$,
$x\notin D$, and $x\notin\mathrm{FV}(e)$

Note that the combined use of rules (Var-GL) and (App) ensures that, in the premise of rule (Rec-GL), the body e of the recursive definition $\mathsf{rec}\,\{x=e\}$ is typed by assigning the same simple type u' to all the occurrences of x. Rule (RecVac-GL) is needed to type *vacuous recursive definitions* (i.e., expressions $\mathsf{rec}\,\{x=e\}$ where $x\notin\mathrm{FV}(e)$).

System $\vdash_{\mathrm{GL}}$ can be seen as derived from system $\vdash_0$ (in Section 3.1) by adapting the second step (see Section 3.2) of the technique illustrated in Section 3. We now adapt the third (see Section 3.3) and the fourth (see Section 3.4) steps to system $\vdash_{\mathrm{GL}}$, obtaining two new type systems that we will call $\vdash^{k}_{\mathrm{GL}}$ and $\vdash^{k,\mathrm{ML}}_{\mathrm{GL}}$, respectively.

Adapting the Third Step. For every $k \geq 1$, let $\vdash^{k}_{\mathrm{GL}}$ be the system obtained from $\vdash_{\mathrm{GL}}$ by replacing rule (Rec-GL) with the rule:

$$(\textsc{Rec-GL-}k)\ \frac{D, x : \langle U_0;\, u_0\rangle \vdash e : \langle U_1, x : u'_1;\, u_1\rangle \ \cdots\ D, x : \langle U_{k-1};\, u_{k-1}\rangle \vdash e : \langle U_k, x : u'_k;\, u_k\rangle}{D \vdash \mathsf{rec}\,\{x = e\} : \langle U_k;\, u_k\rangle}$$

where $x \notin D$,
$\langle U_0;\, u_0\rangle \in \mathsf{B}_{\mathrm{FV}^{\mathrm{GL}}_{D}(\mathsf{rec}\{x=e\})}$,
$(\forall i \in \{1,\ldots,k\})\ \langle U_i, x : u'_i;\, u_i\rangle \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t \mid D, x : \langle U_{i-1};\, u_{i-1}\rangle \vdash e : t\})$,
$\langle U_{k-1};\, u_{k-1}\rangle = \langle U_k;\, u_k\rangle$

Adapting the Fourth Step. For every $k \geq 1$, let $\vdash^{k,\mathrm{ML}}_{\mathrm{GL}}$ the system obtained from $\vdash^{k}_{\mathrm{GL}}$ by adding the rule:

$$(\textsc{Rec-GL-ML})\ \frac{D \vdash e : \langle U, x : u;\, u\rangle}{D \vdash \mathsf{rec}\,\{x = e\} : \langle U;\, u\rangle}$$

where $x \notin D$ and there are no $\langle U_i;\, u_i\rangle$ $(1 \leq i \leq k)$ such that:
$\langle U_0;\, u_0\rangle \in \mathsf{B}_{\mathrm{FV}^{\mathrm{GL}}_{D}(\mathsf{rec}\{x=e\})}$,
$(\forall i \in \{1,\ldots,k\})\ \ \langle U_i, x : u'_i;\, u_i\rangle \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t \mid D, x : \langle U_{i-1};\, u_{i-1}\rangle \vdash e : t\})$,
$\langle U_{k-1};\, u_{k-1}\rangle = \langle U_k;\, u_k\rangle$

5.2 Soundness and Completeness of $\triangleright^{k,\mathrm{ML}}_{\mathrm{GL}}/\triangleright_{\mathrm{GL}}$ w.r.t. $\vdash^{k,\mathrm{ML}}_{\mathrm{GL}}/\vdash_{\mathrm{GL}}$

It is not possible to give a pointwise straightforward correspondence between the structure of the proof tree in system $\vdash^{k,\mathrm{ML}}_{\mathrm{GL}}/\vdash_{\mathrm{GL}}$ and the steps of abstract interpreter $\triangleright^{k,\mathrm{ML}}_{\mathrm{GL}}/\triangleright_{\mathrm{GL}}$ because of the way environments are built. However when $\triangleright^{k,\mathrm{ML}}_{\mathrm{GL}}/\triangleright_{\mathrm{GL}}$ terminates we reach an abstract typing which is isomorphic w.r.t. the corresponding typing in $\vdash^{k,\mathrm{ML}}_{\mathrm{GL}}/\vdash_{\mathrm{GL}}$.

In order to express explicitly the isomorphism between the abstract typing computed by $\triangleright^{k,\mathrm{ML}}_{\mathrm{GL}}/\triangleright_{\mathrm{GL}}$ and the typing in $\vdash^{k,\mathrm{ML}}_{\mathrm{GL}}/\vdash_{\mathrm{GL}}$, we need to introduce some preliminary notations. Let $\mathit{Dom}(H)$ denote the domain of the partial function H. Given H, we write H^1, H^2 as the projection functions (given H, if $x \in \mathit{Dom}(H)$ and $H(x) = (u, \mathbf{s})$, then $H^1(x) = u$ and $H^2(x) = \mathbf{s}$). Also we define $\mathit{Range}(H) = \{\mathrm{FTV}(u) \mid H^1(x) = u, x \in \mathit{Dom}(H)\}$ as the *range of* H.

We can now define the equivalence relations $\approx_H$ and state the correspondence results between $\triangleright^{k,\mathrm{ML}}_{\mathrm{GL}}/\triangleright_{\mathrm{GL}}$ and $\vdash^{k,\mathrm{ML}}_{\mathrm{GL}}/\vdash_{\mathrm{GL}}$.

Definition 18. *Let H be an abstract environment, $\langle U;\, u\rangle$ a typing, and $(u, \mathbf{s})$ an abstract typing. We write $\langle U;\, u\rangle \approx_H (u', \mathbf{s})$ to mean that:*
$u = u'$, $\mathit{Dom}(\mathbf{s}) \subseteq \mathit{Range}(H|_{\mathit{Dom}(U)})$, and (forall $x : u'' \in U$) $\mathbf{s}(H^1(x)) = u''$.

Definition 19. *Let e be an expression, D a typing environment, and H an abstract environment. We write $D \cong H$ to mean that: $\mathit{Dom}(D) \subseteq \mathit{Dom}(H)$ and (for all $x \in \mathit{Dom}(D)$) $x : t \in D$ if and only if $t \approx_H H(x)$.*

Theorem 20 (Soundness and completeness of $\rhd^{k,\mathbf{ML}}_{\mathbf{GL}}$ w.r.t. $\vdash^{k,\mathbf{ML}}_{\mathbf{GL}}$). *Let D be a typing environment and H be an abstract environment with $D \cong H$, then for every $k \geq 1$ and expression e:*
(Soundness) *If $H \rhd^{k}_{\mathrm{GL}} e \Rightarrow (u, \mathbf{s})$, then $D \vdash^{k,\mathrm{ML}}_{\mathrm{GL}} e : t$, with $t \approx_H (u, \mathbf{s})$ and $t \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t' \mid D \vdash^{k,\mathrm{ML}}_{\mathrm{GL}} e : t'\})$.*
(Completeness) *If $D \vdash^{k,\mathrm{ML}}_{\mathrm{GL}} e : t$, then $H \rhd^{k}_{\mathrm{GL}} e \Rightarrow (u, \mathbf{s})$, with $t' \approx_H (u, \mathbf{s})$ for some $t' \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t'' \mid D \vdash^{k,\mathrm{ML}}_{\mathrm{GL}} e : t''\})$.*

Theorem 21 (Soundness and completeness of $\rhd_{\mathbf{GL}}$ w.r.t. $\vdash_{\mathbf{GL}}$). *Let D be a typing environment, H be an abstract environment with $D \cong H$, then for every expression e:*
(Soundness) *If $H \rhd_{\mathrm{GL}} e \Rightarrow (u, \mathbf{s})$, then $D \vdash_{\mathrm{GL}} e : t$, with $t \approx_H (u, \mathbf{s})$ and $t \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t' \mid D \vdash_{\mathrm{GL}} e : t'\})$.*
(Completeness) *If $D \vdash_{\mathrm{GL}} e : t$, then $H \rhd_{\mathrm{GL}} e \Rightarrow (u, \mathbf{s})$ with $t' \approx_H (u, \mathbf{s})$ for some $t' \in \mathbf{Min}_{\leq_{\mathbf{spc}}}(\{t'' \mid D \vdash_{\mathrm{GL}} e : t''\})$.*

5.3 System $\vdash'_{\mathbf{GL}}$: A Simpler Characterization of $\rhd_{\mathbf{GL}}$

Let $\vdash'_{\mathrm{GL}}$ be the system obtained from system $\vdash_0$ in Fig. 1 by removing rule (Spc) and adding the following rules:

$$(\textsc{Spc}')\ \frac{\vdash e : \langle U;\, u\rangle}{\vdash e : \langle U;\, \mathbf{s}(u)\rangle} \quad \text{where } \mathbf{Dom}(\mathbf{s}) = \mathrm{FTV}(u) - \mathrm{FTV}(U)$$

$$(\textsc{Rec-GL}')\ \frac{\vdash e : \langle U, x : \mathbf{s}(u);\, u\rangle}{\vdash \mathsf{rec}\,\{x = e\} : \langle U;\, u\rangle} \quad \text{where } \mathbf{Dom}(\mathbf{s}) = \mathrm{FTV}(u) - \mathrm{FTV}(U)$$

$$(\textsc{RecVac-GL}')\ \frac{\vdash e : \langle U;\, u\rangle}{\vdash \mathsf{rec}\,\{x = e\} : \langle U;\, u\rangle} \quad \text{where } x \notin \mathrm{FV}(e)$$

Note that rule (Spc′) is a restriction of rule (Spc) (rule (Spc) is admissible). The following theorem shows that system $\vdash'_{\mathrm{GL}}$ has the same expressive power as system $\vdash_{\mathrm{GL}}$ and, therefore, of the non-effective type abstract interpreter $\rhd_{\mathrm{GL}}$.

Theorem 22. $\vdash'_{\mathrm{GL}} e : \langle U;\, u\rangle$ *if and only if* $\emptyset \vdash_{\mathrm{GL}} e : \langle U;\, u\rangle$.

6 Conclusions and Future Work

The stratification of (let-free) Milner-Mycroft typability, given in Section 3, is an adaptation of the results in [5] to the case without intersection types. Note that Theorem 13 and Example 14 are completely new (the analogue of Theorem 13 does not hold in presence of intersection types). We plan to extend the type systems (and the results) in Section 3 to deal with let-expressions by exploiting a general technique (developed in [4]) for extending a type system (without rules for let-expressions) enjoying the principal typing property with a typing rule for let-expressions. The characterization of the expressive power of the type abstract interpreters of [6,7], given in Section 5, has been obtained by exploiting the notion of principal typing (which is at the basis of the technique of [5]). We believe that the notion of principal typing might be useful in other investigations on the relations between program analyses specified via type systems and program analyses specified via abstract interpretation. We don't know whether typability in the type system corresponding to the non-effective abstract interpreter $\rhd_{\mathrm{GL}}$ (i.e., typability in systems $\vdash_{\mathrm{GL}}$ and $\vdash'_{\mathrm{GL}}$) is decidable.

References

1. Coppo, M.: An extended polymorphic type system. In: Dembinski, P. (ed.) MFCS 1980. LNCS, vol. 88, pp. 194–204. Springer, Heidelberg (1980)
2. Cousot, P.: Types as abstract interpretations. In: POPL 1997, pp. 316–331. ACM, New York (1997)
3. Damas, L.M.M., Milner, R.: Principal type schemas for functional programs. In: POPL 1982, pp. 207–212. ACM, New York (1982)
4. Damiani, F.: Rank 2 intersection types for local definitions and conditional expressions. ACM Trans. Prog. Lang. Syst. 25(4), 401–451 (2003)
5. Damiani, F.: Rank 2 Intersection for Recursive Definitions. Fundam. Inform. 77(4), 451–488 (2007); (Special Issue on TLCA 2005)
6. Gori, R., Levi, G.: An experiment in type inference and verification by abstract interpretation. In: Cortesi, A. (ed.) VMCAI 2002. LNCS, vol. 2294, pp. 225–239. Springer, Heidelberg (2002)
7. Gori, R., Levi, G.: Properties of a type abstract interpreter. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 132–145. Springer, Heidelberg (2002)
8. Henglein, F.: Type inference with polymorphic recursion. ACM Trans. Prog. Lang. Syst. 15(2), 253–289 (1993)
9. Hindley, R.: Basic Simple Type Theory. Cambridge Tracts in Theoretical Computer Science, vol. 42. Cambridge University Press, London (1997)
10. Jim, T.: What are principal typings and what are they good for?. In: POPL 1996, pp. 42–53. ACM, New York (1996)
11. Kfoury, A.J., Pericas-Geertsen, S.M.: Type inference for recursive definitions. In: LICS 1999, pp. 119–128. IEEE, Los Alamitos (1999)
12. Kfoury, A.J., Tiuryn, J., Urzyczyn, P.: Type reconstruction in the presence of polymorphic recursion. ACM Trans. Prog. Lang. Syst. 15(2), 290–311 (1993)
13. Meertens, L.: Incremental polymorphic type checking in B. In: POPL 1983, pp. 265–275. ACM, New York (1983)
14. Monsuez, B.: Polymorphic typing by abstract interpretation. Theoretical Computer Science 652, 217–228 (1992)
15. Monsuez, B.: Polymorphic types and widening operators. In: Cousot, P., Filé, G., Falaschi, M., Rauzy, A. (eds.) WSA 1993. LNCS, vol. 724, pp. 224–281. Springer, Heidelberg (1993)
16. Mycroft, A.: Polymorphic type schemes and recursive definitions. In: Paul, M., Robinet, B. (eds.) Programming 1984. LNCS, vol. 167, pp. 217–228. Springer, Heidelberg (1984)
17. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge (1998)
18. Terauchi, T., Aiken, A.: On typability for polymorphic recursive rank-2 intersection types. In: LICS 2006, pp. 111–122. IEEE, Los Alamitos (2006)
19. Wells, J.B.: The essence of principal typings. In: Widmayer, P., Triguero, F., Morales, R., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP 2002. LNCS, vol. 2380, pp. 913–925. Springer, Heidelberg (2002)

Modal Abstractions of Concurrent Behaviour

Sebastian Nanz, Flemming Nielson, and Hanne Riis Nielson

Informatics and Mathematical Modelling
Technical University of Denmark
{nanz,nielson,riis}@imm.dtu.dk

Abstract. We present a novel algorithm for the automatic construction of modal transition systems as abstractions of concurrent processes. Modal transition systems are recognised as valuable abstractions for model checking because they allow for the deduction of safety as well as liveness properties. However, the issue of effectively creating these abstractions from specification languages such as process algebras is a missing link that prevents their more widespread usage for model checking of concurrent systems. Our algorithm is based on static analysis and uses a lattice of intervals to express simultaneous over- and under-approximations to the set of process actions available in a particular state. We obtain an abstraction that is 3-valued in both states and transitions and that naturally integrates with model checking approaches for modal transition systems.

1 Introduction

Property-preserving abstractions are the topic of intensive research in model checking, as they provide successful techniques for fighting the state-explosion problem. For example, conventional state transition systems can be abstracted by overapproximation, i.e. by adding more transition paths [3], but then validation is limited to safety properties. *Modal transition systems* [11] use instead two kinds of transition relations, one representing necessary behaviour ("*must*") and the other possible behaviour ("*may*"), while the absence of any transition describes behaviour that cannot possibly occur. Using modal transition systems as abstractions therefore allows for the deduction of both safety and liveness properties.

The benefits of using modal abstractions seem to apply in particular to the verification of concurrent systems, as these are frequently infinite-state (thus requiring abstraction) and their correctness statement is often given by a mix of safety and liveness properties. However, the recently suggested practical techniques for automatic generation of modal abstractions [7,8] mainly apply to classical programming languages, and not to process algebras, one of the most widely used specification formalisms for concurrent systems. Having such an abstraction method would, for example, enable abstraction-based verification frameworks for protocol analysis, where a large community is working with process algebraic specifications.

© Springer-Verlag Berlin Heidelberg 2008

In this paper, we develop an algorithm for constructing modal transition systems as abstractions of processes specified in Milner's *Calculus of Communicating Systems (CCS)* [12]. We choose CCS as the basis of the description of our algorithm because, as a seminal process calculus, it allows us to focus on the essentials of our technique, and thus may facilitate the technique's adaption to more expressive formalisms such as the π-calculus [13] and other concurrent languages. Our approach is rooted in a radical generalisation of *Monotone Frameworks* [15], which have been used in classical data flow analysis and comprise a transfer function

$$\mathit{transfer}_\ell(E) = (E \setminus \mathit{kill}_\ell) \cup \mathit{gen}_\ell$$

yielding the analysis information that holds at the next program point, provided that the information E holds for the current program point ℓ, and where kill_ℓ is the information invalidated, and gen_ℓ the new information created at ℓ. While the classical analyses require an imperative program's control flow graph as an input, our algorithm *constructs* a concurrent system's *abstract* control flow graph besides the associated analysis information. We use intervals to simultaneously over- and under-approximate the multiplicity of the actions which can execute in a certain abstract state, and can hence describe the control flow by a modal transition system. The coarseness of this abstraction can be controlled by a parameter of our algorithm.

Related Work. In recent years, the importance of modal transition systems and related approaches has been emphasised in the context of program analysis and model checking [5,2,9]. In particular, a number of abstraction-based verification frameworks have been developed in this context, where abstractions are described in general abstract interpretation terms [5,18] or using predicate abstraction [7,8]. The method in [8] is so far the only one accompanied with an implementation. In contrast to these works, we consider the abstraction of process calculi, and our technique is based on Monotone Frameworks. We also improve on the expressivity of our own related approach at analysing network behaviour [14,17], where we have considered over-approximations only.

The remainder of this paper is organised as follows. In Section 2 we recapitulate the syntax and operational semantics of CCS. Sections 3 and 4 outline the Monotone Framework which is the key to the algorithmic construction of modal abstractions in Section 5. We conclude in Section 6.

2 Communicating Systems

In this section we give a brief review of the syntax and operational semantics of Milner's *Calculus of Communicating Systems (CCS)* [12].

2.1 Syntax

The syntax of CCS *processes* $P \in \mathbf{Proc}$ and *actions* α is given by the following definition, where we presuppose that *names* x can be drawn from some infinite set.

Table 1. Reaction semantics $P \to_{\tilde{\ell}} Q$

$$\tau^{\ell}.P + Q \;\to_{\ell}\; P \qquad (\overline{x}^{\ell_1}.P_1 + Q_1) \mid (x^{\ell_2}.P_2 + Q_2) \;\to_{\ell_2\ell_1}\; P_1 \mid P_2$$

$$\frac{P \to_{\tilde{\ell}} Q}{P \mid P' \to_{\tilde{\ell}} Q \mid P'} \qquad \frac{P \to_{\tilde{\ell}} P'}{\mathsf{new}\, x\, P \to_{\tilde{\ell}} \mathsf{new}\, x\, P'} \qquad \frac{P' \to_{\tilde{\ell}} Q'}{P \to_{\tilde{\ell}} Q} \text{ if } P \equiv P' \text{ and } Q' \equiv Q$$

$$\begin{aligned} P &::= \mathsf{new}\, x\, P \;\mid\; P_1 \mid P_2 \;\mid\; \Sigma_{i \in I} \alpha_i^{\ell_i}.P_i \;\mid\; A \\ \alpha &::= \overline{x} \;\mid\; x \;\mid\; \tau \end{aligned}$$

Here $\mathsf{new}\, x\, P$ introduces a new name x with scope P. Parallel composition is modelled using the construct $P_1 \mid P_2$ whereas choice is expressed using summations of the form $\Sigma_{i \in I} \alpha_i^{\ell_i}.P_i$ where I is a finite index set. If $I = \emptyset$ we write $\mathsf{0}$ for the process. Sums are guarded and in the case of binary sums they are written $\alpha_1^{\ell_1}.P_1 + \alpha_2^{\ell_2}.P_2$. Actions α are annotated with labels $\ell \in \mathbf{Lab}$ serving as pointers into the process; they will be used in the analysis to be presented shortly but have no semantic significance. Complementary actions take the form $\overline{x}$ and x where x is a name; internal actions are denoted by τ. We shall be interested in *programs* of the form

$$\mathsf{let}\ A_1 \triangleq P_1; \cdots ; A_k \triangleq P_k\ \mathsf{in}\ P_0$$

where the processes named $A_1, \ldots, A_k (\in \mathbf{PN})$ are mutually recursively defined and may be used in the main process P_0 as well as in the process bodies $P_1, \ldots, P_k$.

Example. 1 We introduce a simple running example to illustrate the technical developments throughout the paper. Consider the following program:

$$\mathsf{let}\ \ S \triangleq a^1.r^2.S; \quad Q \triangleq \overline{a}^3.(\overline{r}^4.Q \;\; + \tau^5.\overline{r}^6.Q \;\; + \tau^5.Q)\ \ \mathsf{in}\ \ S \mid Q \mid Q$$

The process S models a semaphore which allows for two actions *acquire* and *release*, and then starts over. The process Q tries to acquire the lock and may then do one of three actions before recursing: release the lock immediately; perform an internal action and release the lock; and, perform an internal action but not release the lock.

2.2 Operational Semantics

Following Chapter 4 of [13], we equip the calculus with a reaction semantics and a structural congruence. The *reaction relation* $P \to_{\tilde{\ell}} Q$ is specified in Table 1 and expresses that the process P may evolve in one step into the process Q; we have annotated the arrow with the labels $\tilde{\ell}$ of the actions involved as this will prove useful when expressing the correctness of the analysis. Note that we use $\tilde{\ell}$ to denote both single labels ℓ and label pairs $\ell_2\ell_1$. The reaction relation naturally extends to programs in that only the main process can evolve.

Table 2. Axioms generating the structural congruence $P \equiv Q$

$$P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \qquad P \mid \mathsf{0} \equiv P$$
$$\mathsf{new}\, x\, \mathsf{0} \equiv \mathsf{0} \qquad \mathsf{new}\, x\, \mathsf{new}\, y\, P \equiv \mathsf{new}\, y\, \mathsf{new}\, x\, P$$
$$\mathsf{new}\, x\, (P \mid Q) \equiv P \mid \mathsf{new}\, x\, Q \;\text{ if }\; x \notin \mathit{fn}(P) \qquad A \equiv P \text{ if } A \triangleq P$$

The *structural congruence* $P \equiv Q$ is defined as the least congruence generated from the axioms of Table 2. Here, $\mathit{fn}(P)$ denotes the set of free names of a process P. To make it possible to interpret the analysis result, we require additionally that processes are congruent if they can be obtained from each other by *disciplined alpha-renaming*; this merely amounts to requiring that each name x has a *canonical name* $\lceil x \rceil$ that is preserved by alpha-renaming.

Example. 2 Using the formal semantics we can express steps of the evolution of the main process $S \mid Q \mid Q$ of Example 1 as follows:

$$S \mid Q \mid Q \;\rightarrow_{1\,3}\; r^2.S \mid (\overline{r}^4.Q \;+ \tau^5.\overline{r}^6.Q \;+ \tau^5.Q) \mid Q \;\rightarrow_5\; r^2.S \mid \overline{r}^6.Q \mid Q$$

The notion of canonical names is lifted to actions so $\lceil \alpha \rceil$ will be the *canonical action* corresponding to α. A program is *consistently labelled* if all occurrences of actions α_1^ℓ and α_2^ℓ with the same label ℓ have the same canonical action, that is $\lceil \alpha_1 \rceil = \lceil \alpha_2 \rceil$. A program is *uniquely labelled* if all occurrences of actions have distinct labels; clearly, a uniquely labelled program is consistently labelled.

Example. 3 The program of Example 1 is consistently but not uniquely labelled.

It is easy to see that the property of being consistently labelled is invariant under the structural congruence and that it is preserved by the reaction semantics; this does not hold for the property of being uniquely labelled. In the following we shall assume to have consistently labelled programs; clearly, such labellings can easily be automatically obtained.

3 Exposed Actions

This section introduces the notion of exposed actions which is used to abstractly represent process configurations. An *exposed action* is an action that *may* participate in the next reaction step. For example, the main process $S \mid Q \mid Q$ of Example 1 has one occurrence of a^1 and two occurrences of $\overline{a}^3$ as exposed actions. In general, a process may contain many, and because of recursion, even infinitely many, occurrences of the same action (all identified by the same label) and several of them may be ready to participate in the next reaction.

To capture this we use *extended multisets* $M \in \mathfrak{M}$ where

$$\mathfrak{M} = \mathbf{Lab} \rightarrow \mathbb{N} \cup \{\infty\}$$

Table 3. Exposed actions

$$\begin{aligned}
\mathcal{E}[\![\mathsf{new}\, x\, P]\!]env &= \mathcal{E}[\![P]\!]env \\
\mathcal{E}[\![P \mid Q]\!]env &= \mathcal{E}[\![P]\!]env +_{\mathfrak{M}} \mathcal{E}[\![Q]\!]env \\
\mathcal{E}[\![\Sigma_{i\in I}\alpha_i^{\ell_i}.P_i]\!]env &= \Sigma_{\mathfrak{M}\, i\in I}\bot_{\mathfrak{M}}[\ell_i \mapsto 1] \\
\mathcal{E}[\![A]\!]env &= env(A) \\
\mathcal{E}_\star[\![P]\!] &= \mathcal{E}[\![P]\!]env_{\mathcal{E}} \\
\text{where } \mathcal{F}_{\mathcal{E}}(env) &= [A_1 \mapsto \mathcal{E}[\![P_1]\!]env, \ldots, A_k \mapsto \mathcal{E}[\![P_k]\!]env] \\
\text{and } env_{\bot_{\mathfrak{M}}} &= [A_1 \mapsto \bot_{\mathfrak{M}}, \ldots, A_k \mapsto \bot_{\mathfrak{M}}] \\
\text{and } env_{\mathcal{E}} &= \textstyle\bigsqcup_{j\geq 0}\mathcal{F}_{\mathcal{E}}^j(env_{\bot_{\mathfrak{M}}})
\end{aligned}$$

and the idea is that $M(\ell)$ records the number of occurrences of the label ℓ; there may be a finite number in which case $M(\ell) \in \mathbb{N}$ or an infinite number so that $M(\ell) = \infty$. We say that a label ℓ is in the domain of M, written $\ell \in dom(M)$, if $M(\ell) \neq 0$. We shall equip the set $\mathfrak{M}$ with a partial ordering $\leq_{\mathfrak{M}}$ defined by:

$$M \leq_{\mathfrak{M}} M' \qquad \text{iff} \qquad \forall \ell : \; M(\ell) \leq M'(\ell) \;\vee\; M'(\ell) = \infty$$

The domain $(\mathfrak{M}, \leq_{\mathfrak{M}})$ is a complete lattice with bottom element $\bot_{\mathfrak{M}} = \lambda\ell.\,0$ and top element $\top_{\mathfrak{M}} = \lambda\ell.\,\infty$, and in addition to least upper and greatest lower bound operators, we shall need the operation $+_{\mathfrak{M}}$, which is an extension of ordinary integer addition with the rule that addition of ∞ yields again ∞.

The exposed actions of a process can be computed by an abstraction function

$$\mathcal{E}_\star : \mathbf{Proc} \to \mathfrak{M}.$$

To motivate the definition let us consider the sum of two processes $\alpha_1^{\ell_1}.P_1 + \alpha_2^{\ell_2}.P_2$. Here both of the actions α_1 and α_2 are ready to interact but none of those of P_1 and P_2 are so we shall take:

$$\mathcal{E}_\star[\![\alpha_1^{\ell_1}.P_1 + \alpha_2^{\ell_2}.P_2]\!] = \bot_{\mathfrak{M}}[\ell_1 \mapsto 1] +_{\mathfrak{M}} \bot_{\mathfrak{M}}[\ell_2 \mapsto 1]$$

If the two labels happened to be equal, the overall count would become 2 since we have used pointwise addition $+_{\mathfrak{M}}$. The rule for parallel composition can be motivated similarly.

To handle the general case we shall introduce the function

$$\mathcal{E} : \mathbf{Proc} \to (\mathbf{PN} \to \mathfrak{M}) \to \mathfrak{M}$$

that as an additional parameter takes an environment holding the required information for the process names. The function is defined in Table 3 for arbitrary processes; in the case of sums it generalises the clauses shown above. The clause for the $\mathsf{new}\, x\, P$ construct simply ignores the introduction of the new name and thereby its scope. For process names we simply consult the environment *env*.

This yields the functional $\mathcal{F}_{\mathcal{E}} : (\mathbf{PN} \to \mathfrak{M}) \to (\mathbf{PN} \to \mathfrak{M})$ shown in Table 3. Since the operations involved in its definition are all monotonic we have a monotonic functional defined on a complete lattice and Tarski's fixed point theorem ensures that it has a least fixed point which is denoted $env_{\mathcal{E}}$ in Table 3. Since all processes are finite it follows that $\mathcal{F}_{\mathcal{E}}$ is continuous and hence that the Kleene formulation of the least fixed point is permissible. We can now define the function $\mathcal{E}_\star$ simply as $\mathcal{E}_\star[\![P]\!] = \mathcal{E}[\![P]\!]env_{\mathcal{E}}$.

Example. 4 For the running example of Example 1 we have:

$$\mathcal{E}_\star[\![S \mid Q \mid Q]\!] = \bot_{\mathfrak{M}}[1 \mapsto 1, 3 \mapsto 2]$$

We may consider another process $R \triangleq (\overline{a}^3.0 + \tau^5.\overline{r}^4.0) \mid R$ to illustrate that also an infinite number of actions can be exposed:

$$\mathcal{E}_\star[\![S \mid R]\!] = \bot_{\mathfrak{M}}[1 \mapsto 1, 3 \mapsto \infty, 5 \mapsto \infty]$$

We can show that the exposed actions are invariant under the structural congruence and that they correctly capture the actions that may be involved in the first reaction step:

Lemma 1. *If $P \equiv Q$ then $\mathcal{E}_\star[\![P]\!] = \mathcal{E}_\star[\![Q]\!]$, and if $P \to_{\tilde{\ell}} Q$ then $\tilde{\ell} \in dom(\mathcal{E}_\star[\![P]\!])$.*

The implementation of the function $\mathcal{E}_\star$ is not completely trivial since the lattice $(\mathfrak{M}, \leq_{\mathfrak{M}})$ has infinite ascending chains that might lead to nontermination because of recursion. We can however show that a suitable combination of a k-fold and a $2k$-fold iteration (k being the number of recursive processes) suffices to calculate the analysis result [17].

4 Monotone Frameworks

The abstraction function $\mathcal{E}_\star$ calculates the information about exposed actions for initial processes. This information changes as the process evolves: once an action has executed, some new actions may become exposed and some may cease to be exposed. For example, in process S of Example 1 the action a^1 is initially exposed but, once executed, it will no longer be exposed (we say that it is *killed*) and instead the action r^2 becomes exposed (we say that it is *generated*).

In this section, we present auxiliary functions allowing us to approximate this evolution of exposed actions during process execution. While the set of exposed actions computed by $\mathcal{E}_\star$ is always precise, this is not the case for generated and killed actions. For example, when executing the action τ^5 of Q, we cannot be sure whether $\overline{r}^6$ or $\overline{a}^3$ becomes exposed. Our approach will be to have the auxiliary functions give both an over- and an under-approximation of the multiplicities of exposed actions, and therefore we use interval lattices as an abstract domain.

4.1 Interval Lattices

We are interested in lifting the domain $\mathbb{N} \cup \{\infty\}$ of the mapping $M \in \mathfrak{M}$ to a more approximative domain: while saying earlier that a label ℓ is exposed

$M(\ell) = k$ times in a process, we now want to express that ℓ is exposed *at least* i times and *at most* j times, and the idea will be to use a new mapping N with $N(\ell) = [i, j]$. The following lattice provides the appropriate domain.

The elements of the lattice $(\mathfrak{I}, \leq_{\mathfrak{I}})$ of intervals over $\mathbb{N} \cup \{\infty\}$ are

$$\mathfrak{I} = \{\bot_{\mathfrak{I}}\} \cup \{[i, j] \ : \ i \leq j, \quad i, j \in \mathbb{N} \cup \{\infty\}\}$$

where $\bot_{\mathfrak{I}}$ denotes the empty interval. The partial ordering $\leq_{\mathfrak{I}}$ is defined as

$$I_1 \leq_{\mathfrak{I}} I_2 \ \text{ iff } \ \inf_{\mathfrak{I}}(I_2) \leq \inf_{\mathfrak{I}}(I_1) \wedge \sup_{\mathfrak{I}}(I_1) \leq \sup_{\mathfrak{I}}(I_2)$$

where the supremum and infimum operators on intervals are given by:

$$\inf_{\mathfrak{I}}(I) = \begin{cases} i & \text{if } I = [i, j] \\ \infty & \text{if } I = \bot_{\mathfrak{I}} \end{cases} \qquad \sup_{\mathfrak{I}}(I) = \begin{cases} j & \text{if } I = [i, j] \\ 0 & \text{if } I = \bot_{\mathfrak{I}} \end{cases}$$

Indeed, $(\mathfrak{I}, \leq_{\mathfrak{I}})$ is then a complete lattice with least element $\bot_{\mathfrak{I}}$ and top element $\top_{\mathfrak{I}} = [0, \infty]$, and its least upper bound operator is given by

$$I_1 \sqcup_{\mathfrak{I}} I_2 = \begin{cases} \bot_{\mathfrak{I}} & \text{if } I_1 = I_2 = \bot_{\mathfrak{I}} \\ [\inf(\inf_{\mathfrak{I}}(I_1), \inf_{\mathfrak{I}}(I_2)),\ \sup(\sup_{\mathfrak{I}}(I_1), \sup_{\mathfrak{I}}(I_2))] & \text{otherwise.} \end{cases}$$

Interval Arithmetic. We would like to define arithmetic operations on elements of the lattice $(\mathfrak{I}, \leq_{\mathfrak{I}})$ in order to enable us to add (and subtract) abstract multiplicities of exposed actions in a similar way as we did for $\mathfrak{M}$. For this let us consider for a moment an interval I to be given by a subset $I \subseteq \mathbb{N} \cup \{\infty\}$ with the property that whenever i and j are in I and $i \leq k \leq j$ then k is in I. An operation $\star_{\mathfrak{I}}$ on intervals can then be defined pointwise

$$I \star_{\mathfrak{I}} J = \{i \star j \ : \ i \in I, j \in J\} \cap (\mathbb{N} \cup \{\infty\})$$

where we take $i + \infty = \infty$ and $i - \infty = -\infty$ for $i \in \mathbb{N} \cup \{\infty\}$. It is easy to establish that, for non-empty intervals, the basic operations addition $+_{\mathfrak{I}}$ and subtraction $-_{\mathfrak{I}}$ of interval arithmetic are equivalently expressed by:

$$\begin{aligned} [i_1, i_2] +_{\mathfrak{I}} [j_1, j_2] &= [i_1 + j_1, i_2 + j_2] \\ [i_1, i_2] -_{\mathfrak{I}} [j_1, j_2] &= [i_1 - j_2, i_2 - j_1] \cap (\mathbb{N} \cup \{\infty\}) \end{aligned}$$

Note that the neutral element of addition is given by $\mathbf{0}_{\mathfrak{I}} = [0, 0]$, i.e. $I +_{\mathfrak{I}} \mathbf{0}_{\mathfrak{I}} = I$, whereas the empty interval $\bot_{\mathfrak{I}}$ is an annihilator, i.e. $I +_{\mathfrak{I}} \bot_{\mathfrak{I}} = \bot_{\mathfrak{I}}$. Adapting this definition of interval arithmetic to the case where intervals are formally elements of $\mathfrak{I}$ is straightforward.

Lemma 2. *The operations $+_{\mathfrak{I}}$ and $-_{\mathfrak{I}}$ enjoy the following properties:*

(1) *$+_{\mathfrak{I}}$ and $-_{\mathfrak{I}}$ are monotonic in both arguments.*
(2) *$+_{\mathfrak{I}}$ is associative and commutative, and for $-_{\mathfrak{I}}$ we have that* $\inf_{\mathfrak{I}}(J) \leq \sup_{\mathfrak{I}}(I)$ *implies* $(I -_{\mathfrak{I}} J) +_{\mathfrak{I}} K \leq_{\mathfrak{I}} (I +_{\mathfrak{I}} K) -_{\mathfrak{I}} J$.

Approximative Multisets. Coming back to our original goal of representing over- and under-approximations of the multiplicity of labels, we can now lift $\mathfrak{I}$ to

$$\mathfrak{N} = \mathbf{Lab} \to \mathfrak{I}.$$

Motivated by our application, we call a set $N \in \mathfrak{N}$ an *approximative multiset.* Using pointwise lifting of the interval domain ordering, the domain $(\mathfrak{N}, \leq_{\mathfrak{N}})$ is again a complete lattice. In particular, the least element $\bot_{\mathfrak{N}}$ is given by $\lambda\ell.\, \bot_{\mathfrak{I}}$ and the largest element $\top_{\mathfrak{N}}$ by $\lambda\ell.\, [0, \infty]$. The neutral element of addition is $\mathbf{0}_{\mathfrak{N}} = \lambda\ell.\, \mathbf{0}_{\mathfrak{I}}$ and the results of Lemma 2 hold analogously.

The relationship of $\mathfrak{N}$ with the lattice of extended multisets $\mathfrak{M} = \mathbf{Lab} \to \mathbb{N} \cup \{\infty\}$ can be made explicit using the operation $\lfloor\, .\, \rfloor$ defined by

$$\lfloor M \rfloor(\ell) = [M(\ell),\, M(\ell)]$$

and it is easy to show that this operation preserves addition, i.e. $\lfloor M +_{\mathfrak{M}} N \rfloor = \lfloor M \rfloor +_{\mathfrak{N}} \lfloor N \rfloor$, and is *non-strict* since $\lfloor \bot_{\mathfrak{M}} \rfloor = \mathbf{0}_{\mathfrak{N}}$.

4.2 Generated and Killed Actions

We shall now introduce two functions $\mathcal{G}_\star$ and $\mathcal{K}_\star$ approximating generated and killed actions. The relevant information will be an element of

$$\mathfrak{T} = \mathbf{Lab} \to \mathfrak{N} \quad (= \mathbf{Lab} \to (\mathbf{Lab} \to \mathfrak{I}))$$

and $(\mathfrak{T}, \leq_{\mathfrak{T}})$ is a complete lattice using pointwise lifting of the ordering of $\mathfrak{N}$. The idea is that $\mathcal{G}_\star[\![P]\!](\ell)$ and $\mathcal{K}_\star[\![P]\!](\ell)$ produce approximative multisets of the actions which are newly exposed and no longer exposed, respectively, when the action labelled ℓ executes. In the end we would like to obtain a transfer function of the kind we have hinted at in Section 1, where generated and killed actions will play the role of the gen_ℓ and $kill_\ell$ components.

To motivate the definition let us consider prefixing as expressed in the process $\alpha^\ell.P$. Clearly, once α^ℓ has been executed it will no longer be exposed whereas the actions of $\mathcal{E}_\star[\![P]\!]$ will become exposed. Thus a first suggestion might be to take $\mathcal{G}_\star[\![\alpha^\ell.P]\!](\ell) = \lfloor \mathcal{E}_\star[\![P]\!] \rfloor$, where we note that we need to lift $\mathcal{E}_\star[\![P]\!]$ from $\mathfrak{M}$ to $\mathfrak{N}$; likewise, we might take $\mathcal{K}_\star[\![\alpha^\ell.P]\!](\ell) = \mathbf{0}_{\mathfrak{N}}[\ell \mapsto [1,1]]$ to express that we decrease the count of ℓ by (at least and at most) 1.

However, in the actual definitions in Table 4 and Table 5 we have to cater for the case where the same label may occur several times in a process (as when ℓ is used inside P), and thus have to take the least upper bound with $\mathcal{G}_\star[\![P]\!]$ and $\mathcal{K}_\star[\![P]\!]$ to ensure that they correctly combine the information available about ℓ. For killed actions it is also worth pointing out that the set N in the clause for summations in Table 5 actually equals $\lfloor \mathcal{E}_\star[\![\Sigma_{i \in I} \alpha_i^{\ell_i}.P_i]\!] \rfloor$ reflecting that *all* the exposed actions of the summation are indeed killed once one of them has been selected for the reaction step. The functions

$$\mathcal{G}_\star : \mathbf{Proc} \to \mathfrak{T} \qquad\qquad \mathcal{K}_\star : \mathbf{Proc} \to \mathfrak{T}$$

are defined using auxiliary functions $\mathcal{G}$ and $\mathcal{K}$, following the same pattern and reasoning for well-definedness used when defining $\mathcal{E}_\star$.

Table 4. Approximative multisets of actions generated

$$\begin{aligned}
\mathcal{G}[\![\mathsf{new}\, x\, P]\!]env &= \mathcal{G}[\![P]\!]env\\
\mathcal{G}[\![P \mid Q]\!]env &= \mathcal{G}[\![P]\!]env \sqcup_{\mathfrak{T}} \mathcal{G}[\![Q]\!]env\\
\mathcal{G}[\![\Sigma_{i\in I}\alpha_i^{\ell_i}.P_i]\!]env &= \textstyle\bigsqcup_{\mathfrak{T}\, i\in I}(\bot_{\mathfrak{T}}[\ell_i \mapsto N] \sqcup_{\mathfrak{T}} \mathcal{G}[\![P_i]\!]env)\\
&\quad \text{where } N = \lfloor \mathcal{E}_\star[\![P_i]\!]\rfloor\\
\mathcal{G}[\![A]\!]env &= env(A)\\[1ex]
\mathcal{G}_\star[\![P]\!] &= \mathcal{G}[\![P]\!]env_{\mathcal{G}}\\
\text{where } \mathcal{F}_{\mathcal{G}}(env) &= [A_1 \mapsto \mathcal{G}[\![P_1]\!]env, \cdots, A_k \mapsto \mathcal{G}[\![P_k]\!]env]\\
\text{and } \ env_{\bot_{\mathfrak{T}}} &= [A_1 \mapsto \bot_{\mathfrak{T}}, \cdots, A_k \mapsto \bot_{\mathfrak{T}}]\\
\text{and } \ env_{\mathcal{G}} &= \textstyle\bigsqcup_{j\geq 0} \mathcal{F}_{\mathcal{G}}^j(env_{\bot_{\mathfrak{T}}})
\end{aligned}$$

Example. 5 Continuing Example 1, the generated and killed actions for the main process $S \mid Q \mid Q$ calculate to:

ℓ	$\mathcal{G}_\star[\![S \mid Q \mid Q]\!](\ell)$	$\mathcal{K}_\star[\![S \mid Q \mid Q]\!](\ell)$
1	$\mathbf{0}_{\mathfrak{N}}[2 \mapsto [1,1]]$	$\mathbf{0}_{\mathfrak{N}}[1 \mapsto [1,1]]$
2	$\mathbf{0}_{\mathfrak{N}}[1 \mapsto [1,1]]$	$\mathbf{0}_{\mathfrak{N}}[2 \mapsto [1,1]]$
3	$\mathbf{0}_{\mathfrak{N}}[4 \mapsto [1,1], 5 \mapsto [2,2]]$	$\mathbf{0}_{\mathfrak{N}}[3 \mapsto [1,1]]$
4	$\mathbf{0}_{\mathfrak{N}}[3 \mapsto [1,1]]$	$\mathbf{0}_{\mathfrak{N}}[4 \mapsto [1,1], 5 \mapsto [2,2]]$
5	$\mathbf{0}_{\mathfrak{N}}[3 \mapsto [0,1], 6 \mapsto [0,1]]$	$\mathbf{0}_{\mathfrak{N}}[4 \mapsto [1,1], 5 \mapsto [2,2]]$
6	$\mathbf{0}_{\mathfrak{N}}[3 \mapsto [1,1]]$	$\mathbf{0}_{\mathfrak{N}}[6 \mapsto [1,1]]$

The next result shows that the information calculated by $\mathcal{G}_\star$ and $\mathcal{K}_\star$ is invariant under the structural congruence and that it potentially *decreases* with the evolution of the process:

Lemma 3. *The functions $\mathcal{G}_\star$ and $\mathcal{K}_\star$ enjoy the following properties:*
(1) *If $P \equiv Q$ then $\mathcal{G}_\star[\![P]\!] = \mathcal{G}_\star[\![Q]\!]$, and if $P \rightarrow_{\tilde{\ell}} Q$ then $\mathcal{G}_\star[\![Q]\!] \leq_{\mathfrak{T}} \mathcal{G}_\star[\![P]\!]$.*
(2) *If $P \equiv Q$ then $\mathcal{K}_\star[\![P]\!] = \mathcal{K}_\star[\![Q]\!]$, and if $P \rightarrow_{\tilde{\ell}} Q$ then $\mathcal{K}_\star[\![Q]\!] \leq_{\mathfrak{T}} \mathcal{K}_\star[\![P]\!]$.*

Finally, the following central result states that generated and killed actions can be combined in order to provide safe approximations to the exposed actions of the resulting process of a reaction:

Theorem 4. *If $P \rightarrow_{\tilde{\ell}} Q$ then $\lfloor \mathcal{E}_\star[\![Q]\!]\rfloor \leq_{\mathfrak{N}} (\lfloor \mathcal{E}_\star[\![P]\!]\rfloor -_{\mathfrak{N}} \mathcal{K}_\star[\![P]\!](\tilde{\ell})) +_{\mathfrak{N}} \mathcal{G}_\star[\![P]\!](\tilde{\ell})$.*

4.3 The Transfer Function

Theorem 4 enables us to adapt the generic transfer function of Section 1 to our setting

$$\mathsf{transfer}_{P_0,\tilde{\ell}}(E) = (E -_{\mathfrak{N}} \mathcal{K}_\star[\![P_0]\!](\tilde{\ell})) +_{\mathfrak{N}} \mathcal{G}_\star[\![P_0]\!](\tilde{\ell}),$$

where E is the the approximative multiset of exposed actions available at a particular program point, which in turn is identified by the actions $\tilde{\ell}$ that may be executed. Note that the transfer function can be precomputed as it relies only on the initial program P_0, and the following corollary confirms that in this manner it still provides a safe approximation similar to the one in Theorem 4.

Table 5. Approximative multisets of actions killed

$$\begin{aligned}
\mathcal{K}[\![\mathsf{new}\, x\, P]\!]env &= \mathcal{K}[\![P]\!]env \\
\mathcal{K}[\![P \mid Q]\!]env &= \mathcal{K}[\![P]\!]env \sqcup_{\mathfrak{T}} \mathcal{K}[\![Q]\!]env \\
\mathcal{K}[\![\Sigma_{i\in I}\alpha_i^{\ell_i}.P_i]\!]env &= \textstyle\bigsqcup_{\mathfrak{T}\, i\in I}(\perp_{\mathfrak{T}}[\ell_i \mapsto N] \sqcup_{\mathfrak{T}} \mathcal{K}[\![P_i]\!]env) \\
&\quad \text{where } N = +_{\mathfrak{N}\, j\in I}\mathbf{0}_{\mathfrak{N}}[\ell_j \mapsto [1,1]] \\
\mathcal{K}[\![A]\!]env &= env(A) \\
\mathcal{K}_\star[\![P]\!] &= \mathcal{K}[\![P]\!]env_{\mathcal{K}} \\
\text{where } \mathcal{F}_{\mathcal{K}}(env) &= [A_1 \mapsto \mathcal{K}[\![P_1]\!]env, \cdots, A_k \mapsto \mathcal{K}[\![P_k]\!]env] \\
\text{and } \; env_{\perp_{\mathfrak{T}}} &= [A_1 \mapsto \perp_{\mathfrak{T}}, \cdots, A_k \mapsto \perp_{\mathfrak{T}}] \\
\text{and } \; env_{\mathcal{K}} &= \textstyle\bigsqcup_{j\geq 0}\mathcal{F}_{\mathcal{K}}^j(env_{\perp_{\mathfrak{T}}})
\end{aligned}$$

Corollary 5. *Consider the program* let $A_1 \triangleq P_1; \cdots ; A_k \triangleq P_k$ in P_0 *and assume* $P_0 \rightarrow^* P \rightarrow_{\tilde{\ell}} Q$. *Then* $\lfloor\mathcal{E}_\star[\![Q]\!]\rfloor \leq_{\mathfrak{N}} \mathsf{transfer}_{P_0,\tilde{\ell}}(\lfloor\mathcal{E}_\star[\![P]\!]\rfloor)$.

Proof. This is an immediate consequence of Theorem 4, Lemmas 2, and 3. □

Example. 6 Following up on Examples 2, 4, and 5, we have $S \mid Q \mid Q \rightarrow^* r^2.S \mid \overline{r}^6.Q \mid Q \rightarrow_{2\,6} S \mid Q \mid Q$ and

$$\begin{aligned}
\lfloor\mathcal{E}_\star[\![r^2.S \mid \overline{r}^6.Q \mid Q]\!]\rfloor &= \mathbf{0}_{\mathfrak{N}}[2 \mapsto [1,1], 3 \mapsto [1,1], 6 \mapsto [1,1]] \\
\mathcal{K}_\star[\![S \mid Q \mid Q]\!](2\,6) &= \mathbf{0}_{\mathfrak{N}}[2 \mapsto [1,1], 6 \mapsto [1,1]] \\
\mathcal{G}_\star[\![S \mid Q \mid Q]\!](2\,6) &= \mathbf{0}_{\mathfrak{N}}[1 \mapsto [1,1], 3 \mapsto [1,1]]
\end{aligned}$$

and hence we can confirm Corollary 5 for this example:

$$\lfloor\mathcal{E}_\star[\![S \mid Q \mid Q]\!]\rfloor \leq_{\mathfrak{N}} \mathsf{transfer}_{P_0,2\,6}(\lfloor\mathcal{E}_\star[\![r^2.S \mid \overline{r}^6.Q \mid Q]\!]\rfloor) = \mathbf{0}_{\mathfrak{N}}[1 \mapsto [1,1], 3 \mapsto [2,2]]$$

5 Constructing Modal Abstractions

Given a program let $A_1 \triangleq P_1; \cdots ; A_k \triangleq P_k$ in P_0 we shall now construct a finite abstraction that faithfully reflects the potentially infinite transition system of the program. Since we are interested in being able to derive both safety and liveness properties from the abstraction, we use *modal transition systems* [11] which offer this expressiveness through the use of two transition relations, one representing an over- and the other an under-approximation of process behaviour.

5.1 Modal Transition Systems

Following Larsen and Thomsen [11], we assume a global set of actions *Act*. A *modal transition system (MTS)* is a triple $(\mathsf{Q}, \longrightarrow, \dashrightarrow)$ where Q is a set of states, $\longrightarrow\, \subseteq \mathsf{Q}\times Act\times\mathsf{Q}$ is a transition relation representing required transitions, called *must*, and $\dashrightarrow\, \subseteq \mathsf{Q} \times Act \times \mathsf{Q}$ is a transition relation representing permissible transitions, called *may*. We require that all necessary behaviour is permissible, i.e. that $\longrightarrow\, \subseteq\, \dashrightarrow$ holds.

We intend to use modal transition systems as abstractions of process behaviour. First we define abstract representations of process configurations.

Definition 6 (Representation). *We say that an approximative multiset $E \in \mathfrak{N}$* represents *a process P, written $P \rhd E$, iff $\lfloor \mathcal{E}_\star[\![P]\!] \rfloor \leq_{\mathfrak{N}} E$.*

A *modal abstraction* is a 5-tuple $(\mathsf{Q}, \longrightarrow, \dashrightarrow, q_0, \mathsf{E})$ where $(\mathsf{Q}, \longrightarrow, \dashrightarrow)$ is a finite modal transition system, $q_0 \in \mathsf{Q}$ is an initial state, and $\mathsf{E} : \mathsf{Q} \rightarrow \mathfrak{N}$ is a mapping which associates every state $q \in \mathsf{Q}$ with an approximative multiset $\mathsf{E}[q] \in \mathfrak{N}$.

In the context of a modal abstraction, we thus may say that a state q represents a process P to mean that $P \rhd \mathsf{E}[q]$. Note that the set *Act* mentioned above is a set of labels $\tilde{\ell}$ in our case. The next definition formalises when a modal abstraction correctly describes the permissible and necessary behaviour of a process P_0.

Definition 7 (Faithfulness). *A modal abstraction $(\mathsf{Q}, \longrightarrow, \dashrightarrow, q_0, \mathsf{E})$ is called* faithful *for P_0 if the initial state represents P_0, i.e. $P_0 \rhd \mathsf{E}[q_0]$, and the following conditions hold for all processes P and states $q \in \mathsf{Q}$ with $P_0 \rightarrow^* P$ and $P \rhd \mathsf{E}[q]$:*

(1) *if $P \rightarrow_{\tilde{\ell}} Q$ then there exists a state $q' \in \mathsf{Q}$ such that $Q \rhd \mathsf{E}[q']$ and $q \dashrightarrow_{\tilde{\ell}} q'$*
(2) *if $q \longrightarrow_{\tilde{\ell}} q'$ then there exists a process Q such that $Q \rhd \mathsf{E}[q']$ and $P \rightarrow_{\tilde{\ell}} Q$*

Reasoning about Modal Abstractions. The modal abstraction suggested by Definition 7 is 3-valued in the following sense: the presence of a must edge implies the presence of concrete behaviour; the absence of any edge implies its absence; and, the presence of a sole may edge does not allow for conclusive decisions. It is thus natural to use the machinery of 3-valued logic [10,16] during the algorithmic construction and for reasoning.

In 3-valued approaches the classical set of truth values is extended with a value 1/2 for expressing uncertainty, i.e. $\mathbb{B}_3 = \{0, 1/2, 1\}$ under the partial order $\leq^3$ induced by $0 \leq^3 1/2 \leq^3 1$. Various modal logics, e.g. CTL in [18], have been adapted for 3-valued reasoning about modal transition systems. In these approaches, states are labelled with 3-valued predicates, similar to classical Kripke structures. We can integrate our approach with these developments by using the function L_E instead of the labelling function for predicates, in order to express whether a label ℓ is necessarily, possibly, or not exposed in E:

$$\mathsf{L}_E(\ell) = \begin{cases} 1 & \text{if } E(\ell) = [m, n] \text{ and } m, n > 0 \\ 1/2 & \text{if } E(\ell) = [0, n] \text{ and } n > 0 \\ 0 & \text{if } E(\ell) = [0, 0] \end{cases}$$

In ongoing work we show that *Action Computation Tree Logic (ACTL)* [6], a version of CTL suitable for reasoning about labelled transition systems, can be used to reason about modal abstractions, in a manner similar to our work in the context of (over-approximating) abstractions [14].

Table 6. The worklist algorithm for constructing the modal abstraction

```
1  $\mathsf{Q} := \{q_0\}; \mathsf{E}[q_0] := \lfloor\mathcal{E}_\star[\![P_0]\!]\rfloor; \mathsf{W} := \{q_0\};\ \longrightarrow\ := \emptyset;\ \dashrightarrow\ := \emptyset;$
2  while $\mathsf{W} \neq \emptyset$ do
3    select $q_s$ from $\mathsf{W}$;  $\mathsf{W} := \mathsf{W} \setminus \{q_s\}$;
4    for each $(\tilde{\ell}, b) \in \mathsf{enabled}(\mathsf{E}[q_s])$ do
5      let $E = \mathsf{transfer}_{P_0,\tilde{\ell}}(\mathsf{E}[q_s])$ in
6        if there exists $q \in \mathsf{Q}$ with $H(\mathsf{E}[q]) = H(E)$ then $q_t := q$
7        else select $q_t$ from outside $\mathsf{Q}$;  $\mathsf{Q} := \mathsf{Q} \cup \{q_t\}$;  $\mathsf{E}[q_t] := E$;  $\mathsf{W} := \mathsf{W} \cup \{q_t\}$ fi;
8        if $\neg(E \leq_{\mathfrak{N}} \mathsf{E}[q_t])$ then $\mathsf{E}[q_t] := \mathsf{E}[q_t] \nabla_{\mathfrak{N}} E$;  $\mathsf{W} := \mathsf{W} \cup \{q_t\}$ fi;
9        $\dashrightarrow\ := (\dashrightarrow \setminus \{q_s \dashrightarrow_{\tilde{\ell}} q\ :\ q \in \mathsf{Q}\}) \cup \{q_s \dashrightarrow_{\tilde{\ell}} q_t\}$;
10       $\longrightarrow\ := (\longrightarrow \setminus \{q_s \longrightarrow_{\tilde{\ell}} q\ :\ q \in \mathsf{Q}\}) \cup \{q_s \longrightarrow_{\tilde{\ell}} q_t\ :\ b = 1\}$
```

5.2 The Worklist Algorithm

We now introduce a worklist algorithm, based on the function $\mathsf{transfer}$ of Section 4.3, that produces a faithful modal abstraction for any input process P. The main data structures of the algorithm are: a set Q of the states introduced so far; a table E that specifies for each $q \in \mathsf{Q}$ the associated approximative multiset $\mathsf{E}[q] \in \mathfrak{N}$; a worklist $\mathsf{W} \subseteq \mathsf{Q}$ of states that have yet to be processed; and two sets $\longrightarrow$ and $\dashrightarrow$ defining the current must and may transitions.

Function $\mathsf{enabled}$. The algorithm makes use of an auxiliary function $\mathsf{enabled}(E)$ to compute *enabled actions*. An action is called enabled if it is exposed *and* can execute. This means that we have to take into account that single labels ℓ can execute only if they label a τ-action, and label pairs $\ell_1\ell_2$ only if the associated actions can synchronise.

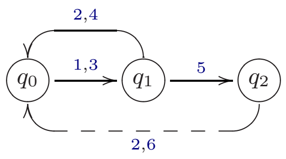
The function $\mathsf{enabled}(E)$ is constructed as an over-approximation but allows us to recover precise results in some cases. To achieve this, $\mathsf{enabled}(E)$ pairs an enabled action $\tilde{\ell}$ with a truth value $b \geq^3 1/2$ to express that $\tilde{\ell}$ is *necessarily* enabled in all processes represented by E (then $b = 1$), or that it may *possibly* be enabled ($b = 1/2$). The computation of $\mathsf{enabled}$ follows the pattern used for $\mathcal{E}_\star$ and is omitted for space reasons; we just state the main result:

Lemma 8. *Suppose* $\mathsf{enabled}$ *was computed for* P_0 *and* $P_0 \rightarrow^* P$.

(1) *If* $P \rightarrow_{\tilde{\ell}} Q$ *then* $(\tilde{\ell}, b) \in \mathsf{enabled}(\lfloor\mathcal{E}_\star[\![P]\!]\rfloor)$ *with* $b \geq^3 1/2$.
(2) *If* $(\tilde{\ell}, 1) \in \mathsf{enabled}(\lfloor\mathcal{E}_\star[\![P]\!]\rfloor)$ *then there exists a process* Q *such that* $P \rightarrow_{\tilde{\ell}} Q$.

Worklist Algorithm. The overall algorithm has the form displayed in Table 6. Several initialisations are performed in line 1. Line 2 contains the classical loop inspecting the contents of the worklist. A source state q_s is selected and removed from the worklist in line 3 and the set of (definite and potential) interactions is constructed using the function call $\mathsf{enabled}(\mathsf{E}[q_s])$ in line 4.

For each pair $(\tilde{\ell}, b) \in \mathsf{enabled}(\mathsf{E}[q_s])$ the procedure call $\mathsf{transfer}_{P_0,\tilde{\ell}}(\mathsf{E}[q_s])$ of line 5 will return an approximative multiset E describing the denotation of the target state. Lines 6-10 are concerned with finding an appropriate target state



q	$\mathsf{E}[q]$
q_0	$\mathbf{0}_{\mathfrak{N}}[1 \mapsto [1,1], 3 \mapsto [2,\infty]]$
q_1	$\mathbf{0}_{\mathfrak{N}}[2 \mapsto [1,1], 3 \mapsto [1,\infty], 4 \mapsto [1,1], 5 \mapsto [2,2]]$
q_2	$\mathbf{0}_{\mathfrak{N}}[2 \mapsto [1,1], 3 \mapsto [1,\infty], 6 \mapsto [0,1]]$

Fig. 1. The modal abstraction for the program of Example 1

q_t and connect it from q_s by updating the transition relations. In line 6 we first decide, using a *granularity function* H, which we will further discuss below, whether one of the existing states can be reused. If this is not possible, a fresh state will be created (line 7).

In line 8 it is checked whether the description $\mathsf{E}[q_t]$ includes the required information E and if not, it is updated and the state is put on the worklist for future processing. The *widening operator* $\nabla_{\mathfrak{N}}$ makes sure to combine the old and the new extended multisets in such a way that termination of the overall algorithm is ensured. We shall return to the definition of $\nabla_{\mathfrak{N}}$ shortly.

The transition relations are updated in lines 9 and 10. In all cases, a may edge between q_s and q_t is added. But only if $b = 1$, i.e. if we are sure that there is a corresponding concrete behaviour due to Lemma 8 (2), we add a must edge. In both cases, old transitions labelled $\tilde{\ell}$ from q_s will be removed, which ensures that we get a modal transition system where no two transitions out of a state have the same label.

Widening Operator. The *widening operator* $\nabla_{\mathfrak{N}} : \mathfrak{N} \times \mathfrak{N} \to \mathfrak{N}$ used in line 8 of Table 6 to combine approximative multisets is defined by:

$$i_1 \nabla i_2 = \begin{cases} i_1 & \text{if } i_2 \leq i_1 \\ i_2 & \text{if } i_1 = 0 \wedge i_2 > 0 \\ \infty & \text{otherwise} \end{cases}$$

$$(N_1 \nabla_{\mathfrak{N}} N_2)(\ell) = \begin{cases} \bot_{\mathfrak{I}} & \text{if } N_1(\ell) = N_2(\ell) = \bot_{\mathfrak{I}} \text{ and otherwise} \\ [\inf(\inf_{\mathfrak{I}} N_1(\ell), \inf_{\mathfrak{I}} N_2(\ell)),\ (\sup_{\mathfrak{I}} N_1(\ell)) \nabla (\sup_{\mathfrak{I}} N_2(\ell))] \end{cases}$$

It will ensure that the chain of values taken by $\mathsf{E}[q_t]$ in line 8 always stabilises after a finite number of steps. We refer to [4,15] for a formal definition of widening and merely establish the correctness of our choice.

Fact 9. *$\nabla_{\mathfrak{N}}$ is a widening operator, in particular $N_1 \sqcup_{\mathfrak{N}} N_2 \leq_{\mathfrak{N}} N_1 \nabla_{\mathfrak{N}} N_2$.*

Granularity Function. We return to the choice of a *granularity function* $H : \mathfrak{N} \to \mathfrak{H}$ to be used in line 6 of Table 6. The function H abstracts $\mathfrak{N}$ by mapping it into some space $\mathfrak{H}$, hence reducing the size of the state space, or making it finite in the first place. We are interested in the granularity function being *finitary*, i.e. mapping into a finite subset $\mathfrak{H}_{\mathsf{fin}} \subseteq \mathfrak{H}$ for all finite sets of labels $\mathbf{Lab}_{\mathsf{fin}} \subseteq \mathbf{Lab}$ as they arise from any concrete program. This ensures termination:

Theorem 10. *If the granularity function H is finitary, then the algorithm of Table 6 always terminates.*

As an example, consider the following family of granularity functions $\mathsf{H}_{i,j}$, where $i, j \in \mathbb{N} \cup \{\infty\}$ and $i \leq j$ (all $\mathsf{H}_{i,j}$ are finitary):

$$\begin{aligned}\mathsf{H}_{i,j}(E) = \{(\ell, [m,n]) &: E(\ell) = [m,n],\, i \leq m,\, n \leq j\} \\ \cup\ \{(\ell, [m,\infty]) &: E(\ell) = [m,n],\, i \leq m,\, n > j\} \\ \cup\ \{(\ell, [0,n]) &: E(\ell) = [m,n],\, i > m,\, n \leq j\} \\ \cup\ \{(\ell, [0,\infty]) &: E(\ell) = [m,n],\, i > m,\, n > j\}\end{aligned}$$

For instance, $\mathsf{H}_{1,1}$ focuses on lower and upper bounds being either 0 or 1, thus abstracting from the actual counts. In this way, $E[\ell \mapsto [1,2]]$ and $E[\ell \mapsto [2,\infty]]$ would both be represented by the same state, while $E[\ell \mapsto [0,2]]$ would not.

Example. 7 For the program of Example 1 and granularity function $\mathsf{H}_{1,1}$, the worklist algorithm produces the modal abstraction shown in Figure 1 (for all must edges, the parallel may edges are omitted). Note that the may edge originating from q_2 describes that after executing τ^5 we cannot be sure whether the lock is released using the synchronisation of the actions r^2 and $\overline{r}^6$, or whether the process is stuck. This is indeed as we would expect.

Correctness. The following result states the correctness of the modal abstraction produced by the worklist algorithm with respect to Definition 7.

Theorem 11. *Suppose that the algorithm of Table 6 terminates and produces a modal abstraction $\mathcal{A}$ for an input process P_0. Then $\mathcal{A}$ is faithful for P_0.*

Implementation. We have implemented our algorithm in OCaml, and have used it to test its performance on a scalable specification of the synchronisation behaviour in the Ingemarsson-Tang-Wong (ITW) protocol [1], a multi-party version of the Diffie-Hellman key agreement protocol. While a presentation of the full example is beyond the scope of this paper, our experimental results suggest practicability of our method, although the prototype implementation uses an explicit, rather than symbolic, encoding of the graph of the modal abstraction. We can also deduce a simple property for the ITW protocol, namely that all synchronisations are forced, since an $\mathsf{H}_{1,1}$-abstraction contains only must edges. In order to be able to check more complex modal properties, we plan to add tool support for (symbolic) model checking of a suitable modal logic (see Section 5.1) in future work.

6 Conclusion

We have shown how to design a static analysis using interval approximations of exposed actions in order to construct modal abstractions of concurrent processes given as terms of a process algebra. A main motivation for this work is the planned integration of our method with model checking, hence obtaining a complete abstraction-based verification framework. It would also be interesting to investigate the use of a relational abstraction, e.g. using polyhedra instead of intervals, in order to increase precision by preserving dependencies between label counts.

References

1. Boyd, C., Mathuria, A.: Protocols for Authentication and Key Establishment. Springer, Heidelberg (2003)
2. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 274–287. Springer, Heidelberg (1999)
3. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Transactions on Programming Languages and Systems 16(5), 1512–1542 (1994)
4. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Principles of Programming Languages (POPL 1979), pp. 269–282. ACM Press, New York (1979)
5. Dams, D., Gerth, R., Grumberg, O.: Abstract interpretation of reactive systems. ACM Transactions on Programming Languages and Systems 19(2), 253–291 (1997)
6. De Nicola, R., Vaandrager, F.W.: Action versus state based logics for transition systems. In: Guessarian, I. (ed.) LITP 1990. LNCS, vol. 469, pp. 407–419. Springer, Heidelberg (1990)
7. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based model checking using modal transition systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 426–440. Springer, Heidelberg (2001)
8. Gurfinkel, A., Chechik, M.: Why waste a perfectly good abstraction? In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, pp. 212–226. Springer, Heidelberg (2006)
9. Huth, M., Jagadeesan, R., Schmidt, D.A.: Modal transition systems: A foundation for three-valued program analysis. In: Sands, D. (ed.) ESOP 2001 and ETAPS 2001. LNCS, vol. 2028, pp. 155–169. Springer, Heidelberg (2001)
10. Kleene, S.C.: Introduction to Metamathematics. Biblioteca Mathematica, vol. 1. North-Holland, Amsterdam (1952)
11. Larsen, K.G., Thomsen, B.: A modal process logic. In: Logic in Computer Science (LICS 1988), pp. 203–210. IEEE Computer Society, Los Alamitos (1988)
12. Milner, R.: Communication and Concurrency. Prentice Hall, Englewood Cliffs (1989)
13. Milner, R.: Communicating and Mobile Systems: The pi-calculus. Cambridge University Press, Cambridge (1999)
14. Nanz, S., Nielson, F., Nielson, H.R.: Topology-dependent abstractions of broadcast networks. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR. LNCS, vol. 4703, pp. 226–240. Springer, Heidelberg (2007)
15. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)
16. Nielson, F., Nielson, H.R., Sagiv, M.: Kleene's logic with equality. Information Processing Letters 80, 131–137 (2001)
17. Nielson, H.R., Nielson, F.: A monotone framework for CCS. Computer Languages, Systems & Structures (under revision) (2006)
18. Shoham, S., Grumberg, O.: A game-based framework for CTL counterexamples and 3-valued abstraction-refinement. ACM Transactions on Computational Logic 9(1) (2007)

Hiding Software Watermarks in Loop Structures

Mila Dalla Preda, Roberto Giacobazzi, and Enrico Visentini

Dipartimento di Informatica, Università di Verona
Strada Le Grazie, 15 – 37134 Verona (Italy)
{mila.dallapreda,roberto.giacobazzi,enrico.visentini}@univr.it

Abstract. In this paper we propose a software watermarking technique based on the fact that different semantic instances might be abstracted in the same syntactic object. Our idea is to hide the watermark in a particular semantic instance and to distribute the corresponding syntactic construct. The extraction process uses a secret key in order to recover the information loss and reconstruct the watermark. In particular, we focus on loops and we base the embedding and extraction algorithm on the semantic understanding of loop-unrolling.

1 Introduction

Nowadays *software piracy*, i.e., the illegal reuse of proprietary code, is a key concern for software developers. *Code obfuscation*, whose aim is to obstruct code decipherment, represents a preventive tool against software piracy: attackers cannot steal what they do not understand [7,8]. Once an attacker goes beyond this defense, *software watermarking* allows the owner of the violated code to prove the ownership of the pirated copies [5,6,14,15]. Software watermarking is a technique for embedding a signature, i.e., an identifier reliably representing the owner, in a program. This allows software developers to prove their ownership by extracting their signature from the pirated copies. A good watermark has to be resilient to distortive attacks and not easy to remove [6].

Most of the existing watermarking techniques target a program feature which can assume *many* configurations, but hide the watermark in just *one* of them. Consider, for example, the watermarking technique [17] that modifies the register allocation: although there are many allocations that suit the program data flow, only one is designated to be the signature and thereby used in the marked program. The same idea applies in [14], where a distinctive permutation of basic blocks is selected among the many possible ones. Both [14] and [17] are *static* techniques, because they affect only the layout of programs. Notice that a statically watermarked program exhibits only the watermark configuration and rules out all the other ones: this may help, rather than hinder, attackers, not to mention the ease of subverting layout while preserving functionality.

Dynamic watermarking techniques exploit configurations that programs assume at runtime, thus allowing many candidate configurations to coexist in the same program. For instance, the path-based technique [4] targets the runtime branching behavior of programs: a program executes different paths on different inputs, but only the special input provides the path that outlines the signature. Likewise, the threading technique [16] yields multi-thread programs in which different configurations arise from how race conditions between threads are resolved; once again, a special input provides

© Springer-Verlag Berlin Heidelberg 2008

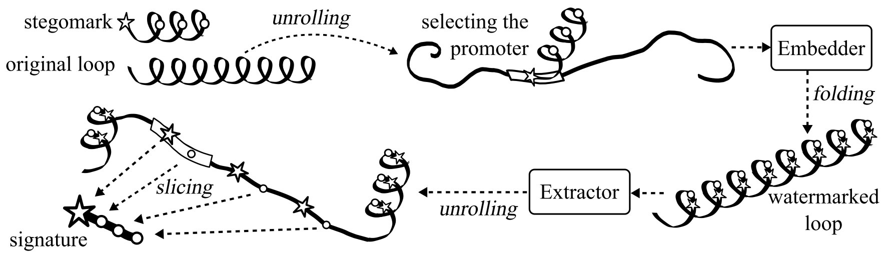


Fig. 1. Watermarking loops with loop-based watermarks

the configuration associated to the signature. Such dynamic techniques are not trivial to thwart: both branching and threading behaviors are tied to functionality, hence their distortion may result in a distortion of functionality. The coexistence of watermarked and unwatermarked configurations within the same program also characterizes the *abstract* watermarking technique [13]. Here a configuration is a parametric abstract domain saying whether a watermark variable w, which is assigned twice and computed through the Horner scheme, is constant or not. Observe that the main point is not the use of the Horner scheme but the fact that w is constant only in the domain parametrized by a key, while other domains consider w to have stochastic behavior [13].

The idea. Contrary to [13], loops are the basic block of the dynamic watermarking technique we propose in this paper. A loop is a programming construct in which a piece of code, called the loop *body*, is executed repeatedly, thus giving rise to sequences of *iterations*. In the proposed technique, *any* subsequence of such sequences is a candidate watermarking configuration. The aim is to embed, in *one* of the subsequences, a *loop-based watermark*, i.e., a watermark that is itself computed iteratively. This is done by enriching the loop body with additional code that yields the signature only within the watermarking subsequence – otherwise it does not produce significant results. Consider for example the program $s := 0$; **for** $i := n$ **to** 50 **do** $s := s + i$ **od**, which performs 50 iterations if $n = 1$. Let the *Beast Software Corporation* have signature 666, computed in 2 iterations by $W := 53$; **for** $i := 17$ **to** 18 **do** $W := W + i^2$ **od**. To watermark the former program, *Beast* moves both $W := 53$ and $W := W + i^2$ in the body of the original loop, thus obtaining program $s := 0$; **for** $i := n$ **to** 50 **do** P_i **od**, in which $P_i \triangleq [W := s - 83;\ s := s + i;\ W := W + i^2]$. Expression $s - 83$ evaluates precisely to 53 only when $n = 1$ and $i = 17$: these are the key values for detecting the watermarking subsequence, which spans two iterations out of 50 (those at $i = 17$ and $i = 18$). At extraction time such a subsequence is made syntactically independent from the native loop: $s := 0$; **for** $i := n$ **to** 16 **do** P_i **od**; P_i; P_{i+1}; **for** $i := 19$ **to** 100 **do** P_i **od**. What is useless for the computation of the signature is then sliced away [19]: $s := 0$; **for** $i := n$ **to** 16 **do** $s := s + i$ **od**; $W := s - 83$; $W := W + i^2$; $W := W + (i+1)^2$. Here, when $n = 1$, W outputs 666. The tool we have used to make the subsequence crop out is *loop-unrolling* [2], a loop transformation that writes out iterations into sequential code, thereby making loop behavior at each iteration syntactically analyzable. As we show in Fig. 1, loop-unrolling is the core of both the embedding and extraction algorithms.

In a native loop L performing N iterations on input I, we can embed a loop-based watermark W requiring $N_W \leq N$ iterations of a code fragment M_W, called *stegomark*. By design, M_W has to get the correct initialization only when it is evaluated in a specific native iteration $\varDelta$, called *promoter*. We designate $\varDelta$ by unrolling L entirely. We establish the dependence that binds M_W to $\varDelta$ through program slicing [19]. Then we fold L and we insert M_W in its body, thus obtaining L_W. For an attacker now unrolling L_W is not of help in determining $\varDelta$ anymore, because M_W appears in every iteration. Moreover, if L_W is contained in program P_W, any loop L' in P_W that includes a fragment of code M' matching the structure of M_W may potentially carry a watermark as well (although $L' \neq L_W$ is highly unlikely to yield a reliable signature). Thus, to retrieve the signature, for each L' we have to: (i) perform a partial unrolling which exposes, if possible, only the subsequence of N_W iterations starting from $\varDelta$; (ii) slice P_W using as criterion the code of M' included in the last iteration of the subsequence; (iii) run the slice on input I and collect the result in the set S of candidate signatures. Finally we have only to identify the signature among the elements of S. Observe that the proposed scheme allows the embedding of *any* kind of loop-based watermarks. In the specific watermarking technique we describe in Sec. 5, the iterative construction of the signature is provided by the evaluation of a polynomial through the Horner scheme as in [13]. We specify programs and their semantics following the syntax and semantics of the simple imperative language described in [12]. Syntactic program transformations, like loop-unrolling and code insertion, are related to their semantic counterpart following the abstract interpretation-based framework of Cousot and Cousot [12].

2 Preliminaries

Notation. Let $\wp(X)$ denote the *powerset* of a set X, namely the set of all subsets of X: $\wp(X) \triangleq \{Y \mid Y \subseteq X\}$. A *poset* is a set X endowed with a partial ordering $\leq_X$, denoted $\langle X, \leq_X \rangle$. Let $\bot_X$ denote, when it exists, the *minimum* of poset X, i.e., $\forall x \in X.\ \bot_X \leq_X x$. An element a is an *upper bound* of X if $\forall x \in X.\ x \leq_X a$. The minimum of the set of upper bounds of X, when it exists, is called the *least upper bound (lub)* of X and it is denoted as $\bigvee X$. A function $f : X \to Y$ from poset X to poset Y is *surjective* when $\forall y \in Y.\ \exists x \in X.\ f(x) = y$. It is $\bot_X$*-strict* when $f(\bot_X) = \bot_Y$. It is *monotonic* if $\forall x, x' \in X.\ x \leq_X x' \implies f(x) \leq_Y f(x')$. It is *additive* if it preserves the lub of every $S \subseteq X$, i.e., $f(\bigvee_X S) = \bigvee_Y f(S)$, where $f(S) \triangleq \{f(x) \mid x \in S\}$. Let $f : X \to X$ be an additive function. A *fixpoint* of f is an element $x \in X$ such that $f(x) = x$. The *least fixpoint* $\mathsf{lfp}^{\leq_X} f$ is the minimum among the fixpoints of f in X.

Abstract Interpretation. In abstract interpretation, any description of program behavior is obtained as an approximation (abstraction) of the most detailed (concrete) program specification available, which is usually a formal semantics [10,11]. Both concrete semantics and abstract behavior are computed on posets: hence there are a *concrete poset* $\langle C, \leq_C \rangle$ and an *abstract poset* $\langle A, \leq_A \rangle$, whose orderings qualitatively model relative precision between elements. When an *abstraction map* $\alpha : C \to A$ and a *concretization map* $\gamma : A \to C$ interrelate the two domains by forming an adjunction, i.e., $\forall c \in C, a \in A.\ \alpha(c) \leq_A a \iff c \leq_C \gamma(a)$, we have a *Galois connection*, denoted $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$. In particular, if α is surjective, we have a *Galois insertion*, denoted

Program Syntax	**Program Semantics**	**Program Abstractions**
Integers $\mathtt{n} \in \mathbb{Z}$	$\mathsf{A}(\mathtt{n})\rho \triangleq n$	$\mathsf{act}(\mathtt{L}\colon \mathtt{A} \to \mathtt{L}';) \triangleq \mathtt{A}$
Variables $\mathtt{Y} \in \mathbb{X}$	$\mathsf{A}(\mathtt{Y})\rho \triangleq \rho(\mathtt{Y})$	$\mathsf{lab}(\mathtt{L}\colon \mathtt{A} \to \mathtt{L}';) \triangleq \mathtt{L}$
Arith. Exps $\mathtt{E} \in \mathbb{E}$,	$\mathsf{A}(\mathtt{E}_1 @ \mathtt{E}_2)\rho \triangleq \mathsf{A}(\mathtt{E}_1)\rho \mathbin{@} \mathsf{A}(\mathtt{E}_2)\rho$	$\mathsf{lab}(\mathtt{P}) \triangleq \bigcup_{\mathtt{C}\in\mathtt{P}}\{\mathsf{lab}(\mathtt{C})\}$
$\mathtt{E} ::= \mathtt{n} \mid \mathtt{Y} \mid \mathtt{E}_1 @ \mathtt{E}_2$	$\mathsf{B}(\mathtt{tt})\rho \triangleq \mathsf{tt}$	$\mathsf{suc}(\mathtt{L}\colon \mathtt{A} \to \mathtt{L}';) \triangleq \mathtt{L}'$
Bool. Exps $\mathtt{B} \in \mathbb{B}$,	$\mathsf{B}(\mathtt{ff})\rho \triangleq \mathsf{ff}$	$\mathsf{suc}(\mathtt{P}) \triangleq \bigcup_{\mathtt{C}\in\mathtt{P}}\{\mathsf{suc}(\mathtt{C})\}$
$\mathtt{B} ::= \mathtt{E}_1 \gtreqless \mathtt{E}_2 \mid \mathtt{B}_1 @ \mathtt{B}_2 \mid \neg\mathtt{B} \mid \mathtt{tt} \mid \mathtt{ff}$	$\mathsf{B}(\neg\mathtt{B})\rho \triangleq \neg\mathsf{B}(\mathtt{B})\rho$	$\mathsf{var}(\mathtt{E}) \triangleq \{\mathtt{Y} \in \mathbb{X} \mid \mathtt{Y} \text{ is in } \mathtt{E}\}$
Actions $\mathtt{A} \in \mathbb{A}$,	$\mathsf{B}(\mathtt{E}_1 \gtreqless \mathtt{E}_2)\rho \triangleq \mathsf{A}(\mathtt{E}_1)\rho \gtreqless \mathsf{A}(\mathtt{E}_2)\rho$	$\mathsf{var}(\mathtt{B}) \triangleq \{\mathtt{Y} \in \mathbb{X} \mid \mathtt{Y} \text{ is in } \mathtt{B}\}$
$\mathtt{A} ::= \mathtt{B} \mid \mathtt{Y} := \mathtt{E} \mid \mathtt{Y} := ?$	$\mathsf{B}(\mathtt{B}_1 @ \mathtt{B}_2)\rho \triangleq \mathsf{B}(\mathtt{B}_1)\rho \mathbin{@} \mathsf{B}(\mathtt{B}_2)\rho$	$\mathsf{var}(\mathtt{Y} := \mathtt{E}) \triangleq \{\mathtt{Y}\} \cup \mathsf{var}(\mathtt{E})$
Symbols $\mathtt{s} \in \mathbb{S}$		$\mathsf{var}(\mathtt{Y} := ?) \triangleq \{\mathtt{Y}\}$
Labels $\mathtt{L} \in \mathbb{L} \triangleq \mathbb{N} \times \mathbb{N} \times \mathbb{S}$	$\mathsf{S}(\mathtt{B})\rho \triangleq \{\rho' \mid \mathsf{B}(\mathtt{B})\rho' = \mathsf{tt} \wedge \rho' = \rho\}$	$\mathsf{var}(\mathtt{C}) \triangleq \mathsf{var}(\mathsf{act}(\mathtt{C}))$
Commands $\mathtt{C} \in \mathbb{C}$,	$\mathsf{S}(\mathtt{Y} := \mathtt{E})\rho \triangleq \{\rho[\mathtt{Y} := \mathsf{A}(\mathtt{E})\rho]\}$	$\mathsf{var}(\mathtt{P}) \triangleq \bigcup_{\mathtt{C}\in\mathtt{P}} \mathsf{var}(\mathtt{C})$
$\mathtt{C} ::= \mathtt{L}\colon \mathtt{A} \to \mathtt{L}';$	$\mathsf{S}(\mathtt{Y} := ?)\rho \triangleq \{\rho' \mid \exists z \in \mathbb{Z}.\ \rho' = \rho[\mathtt{Y} := z]\}$	
Programs $\mathtt{P} \in \mathbb{P} \triangleq \wp(\mathbb{C})$		

Fig. 2. Syntactic and semantic program constructs

$C \xleftrightarrows[\alpha]{\gamma} A$; it can be proved that we always have a Galois insertion whenever α, γ are monotonic, $c \leq_C \gamma(\alpha(c))$ and $\alpha(\gamma(a)) = a$. Given a Galois connection $C \xleftrightarrows[\alpha]{\gamma} A$, a concrete function $f : C \to C$ and an abstract function $f^\sharp : A \to A$, we say that $f^\sharp$ is a *correct* approximation of f in A if $\alpha \circ f \leq_A f^\sharp \circ \alpha$. We let $f^A \triangleq \alpha \circ f \circ \gamma$ denote the *best correct approximation* of f on A. When the correctness condition is strengthened to equality, i.e., when $\alpha \circ f = f^\sharp \circ \alpha$, the abstract function $f^\sharp$ is a *complete* approximation of f on A. When α is $\bot_C$-strict and additive and $f^\sharp$ is complete wrt. f and A, then $\alpha(\mathsf{lfp}^{\leq_C} f) = \mathsf{lfp}^{\leq_A} f^\sharp$, i.e., no loss of information is accumulated in the abstract computation through $f^\sharp$ [1,9]. Then a *fixpoint transfer* can be made from C to A.

Programming Language. We consider the imperative language introduced in [12] (see Fig. 2). Any command $\mathtt{C}$ has the form $\mathtt{L}\colon \mathtt{A} \to \mathtt{L}';$, meaning that $\mathtt{C}$ is referred to through label $\mathtt{L}$, performs action $\mathtt{A}$ and in turn refers to commands with label $\mathtt{L}'$. $\mathtt{A}$ can be either a deterministic ($\mathtt{Y} := \mathtt{E}$) or random assignment ($\mathtt{Y} := ?$), or a boolean test evaluation. A *label* or *entrypoint* $\mathtt{L} \triangleq im\mathtt{s}$ consists of an *index* $i \in \mathbb{N}$, a *memory value* $m \in \mathbb{N}$ and a symbol $\mathtt{s}$ from an alphabet $\mathbb{S}$: whenever $i, m > 0$, we have that $\mathtt{C}$ is the m-th copy of a native command $\overline{\mathtt{C}}$ at entrypoint $00\mathtt{s}$ and $\mathtt{C}$ is also member of the i-th unrolled loop (see Sect. 4). A *program* $\mathtt{P}$ is a possibly infinite set of commands[1] whose execution starts at entrypoints in $\mathfrak{L}(\mathtt{P}) \subseteq \mathsf{lab}(\mathtt{P})$. Program variables in $\mathtt{P}$ take their values in an *environment* $\rho \in \mathfrak{E}(\mathtt{P})$, which is a mapping from $\mathsf{var}(\mathtt{P}) = \mathsf{dom}\,[\rho]$ to $\mathbb{Z} \cup \{\mho\}$, where $\mho \notin \mathbb{Z}$ is the undefined value. When the domain of ρ is not relevant, we can write $\rho \in \mathfrak{E}$. As shown in Fig. 2, we use functions $\mathsf{A}(\mathtt{E}) : \mathfrak{E}(\mathtt{P}) \to \mathbb{Z} \cup \{\mho\}$ and $\mathsf{B}(\mathtt{B}) : \mathfrak{E}(\mathtt{P}) \to \{\mathsf{tt}, \mathsf{ff}, \mho\}$ to evaluate arithmetic ($\mathtt{E}$) or boolean ($\mathtt{B}$) expressions of $\mathtt{P}$; evaluation propagates $\mho$ from subexpressions to superexpressions. We also use function $\mathsf{S}(\mathtt{A}) : \mathfrak{E}(\mathtt{P}) \to \wp(\mathfrak{E}(\mathtt{P}))$, which evaluates action $\mathtt{A}$ by returning the set of environments $\mathtt{A}$ generates when executed. A *state* $s = \langle \rho, \mathtt{C} \rangle$ pairs an environment $\rho \in \mathfrak{E}(\mathtt{P})$ with a command $\mathtt{C} \in \mathtt{P}$. The set of states resulting from the execution of $\mathtt{C}$ in ρ is $\mathsf{S}(\langle \rho, \mathtt{C} \rangle) \triangleq \{\langle \rho', \mathtt{C}' \rangle \mid \mathtt{C}' \in \mathtt{P} \wedge \rho' \in \mathsf{S}(\mathsf{act}(\mathtt{C}))\rho \wedge \mathsf{suc}(\mathtt{C}) = \mathsf{lab}(\mathtt{C}')\}$; relation S models

[1] Here we follow [12] and consider programs as possibly infinite sequences of commands.

the *transition* between states. From the set $\mathfrak{L}(\mathtt{P}) \subseteq \mathsf{lab}(\mathtt{P})$ of the initial entrypoints of $\mathtt{P}$, we can define the set $\mathfrak{I}(\mathtt{P}) \triangleq \{\langle\rho, \mathtt{C}\rangle \mid \rho \in \mathfrak{E}(\mathtt{P}) \wedge \mathtt{C} \in \mathtt{P} \wedge \mathsf{lab}(\mathtt{C}) \in \mathfrak{L}(\mathtt{P})\}$ of the initial states of $\mathtt{P}$. *Trace semantics* $\mathsf{S}(\mathtt{P}) \triangleq \mathsf{lfp}^{\subseteq} \mathsf{F}(\mathtt{P})$ is the least fixpoint of an operator $\mathsf{F}(\mathtt{P})\mathcal{T} \triangleq \mathfrak{I}(\mathtt{P}) \cup \{\sigma s s' \mid \sigma s \in \mathcal{T} \wedge s' \in \mathsf{S}(s)\}$. Each *finite partial trace* $\sigma \in \mathsf{S}(\mathtt{P}) \subseteq \mathfrak{D}$ records a finite partial execution of $\mathtt{P}$. We let $\mathfrak{D}$ be the set of the finite partial traces of all programs and σ_j be the $(j+1)$-th state of σ. A set $\mathcal{T} \in \wp(\mathfrak{D})$ of traces can be abstracted by collecting only the commands executed along the traces [12]. Thus $\mathbb{p}(\mathcal{T}) \triangleq \{\mathtt{C} \mid \exists\sigma \in \mathcal{T}.\ \exists j \in [0, |\sigma|).\ \exists\rho \in \mathfrak{E}.\ \sigma_j = \langle\rho, \mathtt{C}\rangle\}$ induces a Galois insertion $\langle\wp(\mathfrak{D}), \subseteq\rangle \overset{\mathsf{S}}{\underset{\mathbb{p}}{\leftrightarrows}} \langle\mathbb{P}/_{\equiv}, \sqsubseteq\rangle$ which interprets programs as an abstraction of their (trace) semantics. In the abstract domain of programs, $\mathtt{P}$ and $\mathtt{Q}$ collapse ($\mathtt{P} \equiv \mathtt{Q}$) iff $\mathsf{S}(\mathtt{P}) = \mathsf{S}(\mathtt{Q})$ up to semantic equivalences between actions (for instance, $\mathtt{Y} := 6$ is semantically equivalent to $\mathtt{Y} := 2 \times 3$). The *syntactic refinement* $\mathtt{P} \sqsubseteq \mathtt{Q}$ holds iff $\mathsf{S}(\mathtt{P}) \subseteq \mathsf{S}(\mathtt{Q})$ up to semantic equivalences between actions.

Principle of Program Transformation. Any syntactic program transformer $\mathbb{t}$, altering the code of $\mathtt{P}$ and returning new program $\mathtt{P}'$, induces a corresponding semantic transformer t turning $\mathsf{S}(\mathtt{P})$ into $\mathsf{S}(\mathtt{P}')$ [12]. If we let $t(\mathcal{T}) \triangleq \{t(\sigma) \mid \sigma \in \mathcal{T}\}$, then t induces a Galois connection $\langle\wp(\mathfrak{D}), \subseteq\rangle \overset{\gamma_t}{\underset{t}{\leftrightarrows}} \langle\wp(\mathfrak{D}), \subseteq\rangle$, where t acts as an abstraction. By composing the two Galois connection introduced so far, we can derive a similar notion about $\mathbb{t}$, i.e., $\langle\mathbb{P}/_{\equiv}, \sqsubseteq\rangle \overset{\gamma_{\mathbb{t}}}{\underset{\mathbb{t}}{\leftrightarrows}} \langle\mathbb{P}/_{\equiv}, \sqsubseteq\rangle$ [12]. This kind of composition allows us to derive $\mathbb{t}$ as the best correct approximation of semantic transformer t, i.e., $\mathbb{t} \sqsupseteq \mathbb{p} \circ t \circ \mathsf{S}$. In particular, when the transformation is decidable, we have $\mathbb{t} \equiv \mathbb{p} \circ t \circ \mathsf{S}$. The systematic design of $\mathbb{t}$ from t takes advantage of fixpoint transfers [12]. In the following we consider only decidable transformations, such as loop-unrolling and assignment-insertion. Hence, we derive $\mathbb{t} \triangleq \mathsf{lfp}^{\sqsubseteq} \mathbb{F}^t$ in fixpoint form by combining $\mathbb{t} \equiv \mathbb{p} \circ t \circ \mathsf{S}$ with the following equivalence: $\mathbb{p} \circ t \circ \mathsf{S} = \mathbb{p} \circ t \circ \mathsf{lfp}^{\subseteq} \mathsf{F} = \mathbb{p} \circ \mathsf{lfp}^{\subseteq} \mathsf{F}^t \equiv \mathsf{lfp}^{\sqsubseteq} \mathbb{F}^t$. Notice that the first equality follows by definition of S; the other ones hold only if operators $\mathsf{F}^t : \wp(\mathfrak{D}) \to \wp(\mathfrak{D})$ and $\mathbb{F}^t : \mathbb{P}/_{\equiv} \to \mathbb{P}/_{\equiv}$ are designed to fit the requirements of the fixpoint transfers applied within Galois connections $\langle\wp(\mathfrak{D}), \subseteq\rangle \overset{\gamma_t}{\underset{t}{\leftrightarrows}} \langle\wp(\mathfrak{D}), \subseteq\rangle$ and $\langle\wp(\mathfrak{D}), \subseteq\rangle \overset{\mathsf{S}}{\underset{\mathbb{p}}{\leftrightarrows}} \langle\mathbb{P}/_{\equiv}, \sqsubseteq\rangle$ respectively. The correctness of $\mathbb{t}$ is formalized through some *observational abstraction* $\alpha_{\mathcal{O}}$ such that $\langle\wp(\mathfrak{D}), \subseteq\rangle \overset{\gamma_{\mathcal{O}}}{\underset{\alpha_{\mathcal{O}}}{\leftrightarrows}} \langle\mathcal{D}_{\mathcal{O}}, \sqsubseteq_{\mathcal{O}}\rangle$. Transformer $\mathbb{t}$ is correct wrt. $\alpha_{\mathcal{O}}$ if and only if for every program $\mathtt{P} \in \mathbb{P}$, $\alpha_{\mathcal{O}}(\mathsf{S}(\mathtt{P})) = \alpha_{\mathcal{O}}(t(\mathsf{S}(\mathtt{P})))$ [12].

3 Assignment-Insertion

Let us define in the framework described above the transformation of assignment-insertion that we exploit for the embedding of M_W. Suppose we wish to insert, at entrypoint $\mathtt{J} \in \mathsf{lab}(\mathtt{P})$, an assignment $\mathtt{W} := \mathtt{E}$; we use $\mathtt{J}, \mathtt{J}' \ldots$ to denote labels targeted for insertion and $\mathtt{L}, \mathtt{L}' \ldots$ to denote other labels. Syntactically, the solution is straightforward (see Fig. 3): we modify in $\mathtt{P}$ every command referring to $\mathtt{J}$ so that now it refers to a new $\underline{\mathtt{J}} \notin \mathsf{lab}(\mathtt{P})$, then we insert in $\mathtt{P}$ a new command $\underline{\mathtt{C}} \triangleq \underline{\mathtt{J}} \colon \mathtt{W} := \mathtt{E} \to \mathtt{J};$,

```
        // Original program P
00e     Z := 0;
00f     X := 0;
        // Native for-loop F
00g     $\neg(X < \mathcal{I}_0) \to$ end;
00g     $X < \mathcal{I}_0 \to$ 00h;
00h       Y := $\mathsf{Fib}(X + \mathcal{I}_1)$;
00v       Z := $Z + Y \times Y$;
00i       X := $X + 1 \to$ 00g;
```

```
          // Watermarked program P'
   00e    Z := 0;
   00f    X := 0;
          // for-loop F'
   00g    $\neg(X < \mathcal{I}_0) \to$ end;
   00g    $X < \mathcal{I}_0 \to$ 00h;
   00h      Y := $\mathsf{Fib}(X + \mathcal{I}_1)$;
→  00$\underline{\text{v}}$      W := $(215 - Y) \times 259$;
   00v      Z := $Z + Y \times Y$;
→  00$\underline{\text{i}}$      W := $14 \times W + 245760$;
   00i      X := $X + 1 \to$ 00g;
```

Assignment-insertion turns P into P′. Then P′ yields program Q provided that its for-loop F′ is unrolled with $\mathbf{u} = \langle 1, 3 \rangle$ and $\boldsymbol{\ell} = \langle 5, 2 \rangle$. The entrypoint of each program is 00e. The new assignments carry the watermark of signature $\mathfrak{s} = 120736$. To extract $\mathfrak{s}$, we need to run on P′ the algorithm described in Sec. 5. The algorithm computes Q and try to detect a candidate assignment (★). The copies (⋆) of that assignment are discarded. In resulting program Q′ it determines (→ ∘) backward-slicing criterion C and computes slicing S. On the key input, the commands in S yield signature $\mathfrak{s}$ embedded in P′.

```
                 // Program Q
                 // generated at extraction time
         00e     Z := 0;
      S  00f     X := 0;
                 // unrolled for-loop F_1 ($u_1 = 1$)
      S  10g     $\neg(X < \mathcal{I}_0 - 5) \to$ 20g;
      S  10g     $X < \mathcal{I}_0 - 5 \to$ 10h;
         10h       Y := $\mathsf{Fib}(X + \mathcal{I}_1)$;
         10$\underline{\text{v}}$       W := $(215 - Y) \times 259$;
         10v       Z := $Z + Y \times Y$;
         10$\underline{\text{i}}$       W := $14 \times W + 245760$;
      S  10i       X := $X + 1 \to$ 10g;
                 // unrolled for-loop F_2 ($u_2 = 3$)
      S  20g     $\neg(X < \mathcal{I}_0 - 4) \to$ 00g;
      S  20g     $X < \mathcal{I}_0 - 4 \to$ 20h;
      S  20h       Y := $\mathsf{Fib}(X + \mathcal{I}_1)$;
★     S  20$\underline{\text{v}}$       W := $(215 - Y) \times 259$;
         20v       Z := $Z + Y \times Y$;
∘     S  20$\underline{\text{i}}$       W := $14 \times W + 245760$;
         20i       tt;
         21g       tt;
         21h       Y := $\mathsf{Fib}((X + 1) + \mathcal{I}_1)$;
⋆        21$\underline{\text{v}}$       W := $(215 - Y) \times 259$;
         21v       Z := $Z + Y \times Y$;
∘     S  21$\underline{\text{i}}$       W := $14 \times W + 245760$;
         21i       tt;
         22g       tt;
         22h       Y := $\mathsf{Fib}((X + 2) + \mathcal{I}_1)$;
⋆        22$\underline{\text{v}}$       W := $(215 - Y) \times 259$;
         22v       Z := $Z + Y \times Y$;
→ ∘   S  22$\underline{\text{i}}$       W := $14 \times W + 245760$;
      S  22i       X := $X + 3 \to$ 20g;
                 // for-loop F of program P'
         00g     [...]
```

Fig. 3. Three programs

obtaining P′. In case $W \in \mathsf{var}(P)$, however, $\mathsf{S}(P')$ may differ a lot from $\mathsf{S}(P)$, as the new value of W may alter deeply the evaluations of subsequent boolean conditions and control flow of P. To prevent these major changes, we might restore the value of W before W is used again. Alternatively, we just exploit a variable W that is either *fresh* wrt. P, namely $W \notin \mathsf{var}(P)$, or *dead* wrt. entrypoint J targeted for the insertion, i.e., commands executed after the reaching of J must not define or use W any longer. Under this hypothesis, $\langle \rho_0, \mathtt{L}\colon \mathtt{A}_0 \to \mathtt{J};\rangle\langle \rho_1, \mathtt{J}\colon \mathtt{A}_1 \to \mathtt{L}';\rangle \in \mathsf{S}(P)$ can be transformed into $\langle \varphi_0, \mathtt{L}\colon \mathtt{A}_0 \to \underline{\mathtt{J}};\rangle\langle \varphi_1, \underline{\mathtt{C}}\rangle\langle \varphi_1', \mathtt{J}\colon \mathtt{A}_1 \to \mathtt{L}';\rangle \in \mathsf{S}(P')$. We let φ_0, φ_1 be ρ_0, ρ_1 enriched with $W \mapsto \mho$ in case W is fresh. On the other side, we let $\varphi_1' \triangleq \rho_1 \oplus \{W \mapsto \mathsf{A}(E)\rho_1\}$, meaning that W has to belong to $\mathsf{dom}\,[\varphi_1']$ and has to take value $\mathsf{A}(E)\rho_1$. We say that φ_1' is an enhancement of ρ_1. In general, given $\rho \in \mathfrak{E}$, its enhancement $\rho \oplus \omega \triangleq (\rho \setminus \omega) \cup \omega$

$$t^{\text{in}}(\varepsilon\langle\rho,\mathtt{C}\rangle) \triangleq t^{\text{in}}(\langle\rho,\mathtt{C}\rangle, \{\mathtt{W_J} \leftarrow \mho \mid \mathtt{W_J} \in \mathcal{W} \wedge \mathtt{W_J} \notin \mathsf{dom}\,[\rho]\})$$

$$t^{\text{in}}(\sigma'\langle\rho',\mathtt{C}'\rangle\langle\rho,\mathtt{C}\rangle) \triangleq \textbf{let}\ \overline{\sigma}\langle\overline{\rho},\overline{\mathtt{C}}\rangle = t^{\text{in}}(\sigma'\langle\rho',\mathtt{C}'\rangle)\ \textbf{in}\ \overline{\sigma}\langle\overline{\rho},\overline{\mathtt{C}}\rangle t^{\text{in}}(\langle\rho,\mathtt{C}\rangle, \overline{\rho} \text{ restricted to domain } \mathcal{W})$$

$$t^{\text{in}}(\langle\rho,\mathtt{C}\rangle,\omega) \triangleq \textbf{let}\ \varphi = \rho \oplus \omega\ \textbf{in match}\ t^{\text{in}}(\mathtt{C})\ \textbf{with}$$

$$\quad \mathtt{J\colon W_J := E_J \to L';C'} \longmapsto \langle\varphi, \mathtt{J\colon W_J := E_J \to L';}\rangle\langle\varphi[\mathtt{W_J} := \mathsf{A}(\mathtt{E_J})\omega], \mathtt{C}'\rangle$$

$$\quad \mathtt{C}' \longmapsto \langle\varphi, \mathtt{C}'\rangle$$

$$t^{\text{in}}(\mathtt{J\colon A \to J';}) \triangleq \underline{\mathtt{J}}\mathtt{\colon W_J := E_J \to J;J\colon A \to }\underline{\mathtt{J}}\mathtt{';} \qquad t^{\text{in}}(\mathtt{L\colon A \to J';}) \triangleq \mathtt{L\colon A \to }\underline{\mathtt{J}}\mathtt{';}$$

$$t^{\text{in}}(\mathtt{J\colon A \to L';}) \triangleq \underline{\mathtt{J}}\mathtt{\colon W_J := E_J \to J;J\colon A \to L';} \qquad t^{\text{in}}(\mathtt{L\colon A \to L';}) \triangleq \mathtt{L\colon A \to L';}$$

Fig. 4. Semantic assignment-insertion

augments ρ with the mappings in $\omega \in \mathfrak{E}$, overwriting $\mathsf{A}(\mathtt{W})\rho$ with $\mathsf{A}(\mathtt{W})\omega$ in case there is a clash on $\mathtt{W}$. Due to the triviality of the trace transformation, observational abstraction $\alpha^{\text{in}}_{\mathcal{O}}$ for assignment-insertion needs only to discard the inserted states and return the resulting sequence, expunged from environments:

$$\alpha^{\text{in}}_{\mathcal{O}}(\sigma) \triangleq \lambda j.\, \alpha^{\text{in}}_{\mathcal{O}}(\sigma_j) \qquad \alpha^{\text{in}}_{\mathcal{O}}(im\mathtt{s\colon A \to L';}) \triangleq im\mathtt{s\colon A \to L';}$$

$$\alpha^{\text{in}}_{\mathcal{O}}(\langle\rho,\mathtt{C}\rangle) \triangleq \alpha^{\text{in}}_{\mathcal{O}}(\mathtt{C}) \qquad \alpha^{\text{in}}_{\mathcal{O}}(im\underline{\mathtt{s}}\mathtt{\colon A \to L';}) \triangleq \varepsilon\ .$$

Semantic transformation t^{in} for assignment-insertion, shown in Fig. 4, scans the traces of $\mathtt{P}$ state by state, performing different insertions. Each time it finds entrypoint $\mathtt{J}$, it inserts correspondent assignment $\mathtt{W_J} := \mathtt{E_J}$. We have that each $\mathtt{J}$ is a target label in $\mathtt{P}$, each $\mathtt{W_J}$ is a variable from a set $\mathcal{W}$ and each $\mathtt{E_J}$ is such that $\mathsf{var}(\mathtt{E_J}) \subseteq \mathsf{var}(\mathtt{P}) \cup \mathcal{W}$. The algorithm also enhances the environments of the trace, replacing each ρ with $\rho \oplus \omega$. To this purpose, it maintains a special environment ω which changes dynamically from state to state. At the beginning, it tracks only the fresh variables in $\mathcal{W}$, mapping each one to $\mho$. In the following, after a state has been transformed, it tracks all variables in $\mathcal{W}$, deriving their values from the (enhanced) environment of the transformed state. Since each $\mathtt{W_J} \in \mathcal{W}$ is dead at entrypoint $\mathtt{J}$, the enhancement of ρ influence neither the evaluation of arithmetic and boolean expressions, nor the control flow, as desired. We can formally prove this fact by taking advantage of $\alpha^{\text{in}}_{\mathcal{O}}$.

Proposition 1. *Let* $\mathtt{P}$ *be a program and* $\sigma \in \mathsf{S}(\mathtt{P})$. *Then* $t^{\text{in}}(\sigma)$ *is a trace and* $\alpha^{\text{in}}_{\mathcal{O}}(t^{\text{in}}(\sigma)) = \alpha^{\text{in}}_{\mathcal{O}}(\sigma)$. *Furthermore* $\mathbb{p} \circ (t^{\text{in}} \circ \mathsf{F}) = \mathbb{F}^{\text{in}} \circ \mathbb{p}$.

The fixpoint transfer inside the proposition allows us to derive from t^{in} a syntactic algorithm $\mathbb{t}^{\text{in}}$ for assignment-insertion. We express it as follows:

$$\textsc{Init}^{\text{in}}(\mathtt{P}) \triangleq \{\mathtt{C}' \mid \exists \mathtt{C} \in \mathtt{P}.\, \mathsf{lab}(\mathtt{C}) \in \mathfrak{L}(\mathtt{P}) \ \wedge \mathtt{C}' \text{ is in } t^{\text{in}}(\mathtt{C})\}$$

$$\textsc{Next}^{\text{in}}(\mathtt{P})(\mathtt{Q}) \triangleq \{\mathtt{C}' \mid \exists \mathtt{C} \in \mathtt{P}.\, \exists \mathtt{D} \in \mathtt{Q}.\, \mathtt{C}' \text{ is in } t^{\text{in}}(\mathtt{C}) \wedge$$
$$\mathsf{lab}(\mathtt{C}) = im\mathtt{s} \wedge (\mathsf{suc}(\mathtt{D}) = im\mathtt{s} \vee \mathsf{suc}(\mathtt{D}) = im\underline{\mathtt{s}})\}$$

$$\textsc{Iter}^{\text{in}}(\mathtt{P})(\mathtt{Q}) \triangleq \ \textbf{let}\ \mathtt{Q}' = \mathtt{Q} \cup \textsc{Next}^{\text{in}}(\mathtt{P})(\mathtt{Q})\ \textbf{in if}\ \mathtt{Q}' = \mathtt{Q}\ \ \textbf{then}\ \mathtt{Q}'\ \textbf{else}\ \textsc{Iter}^{\text{in}}(\mathtt{P})(\mathtt{Q}')\ \textbf{fi}.$$

4 Loop-Unrolling

The easiest looping constructs to unroll are `for`-loops. Program $\mathtt{P}$ of Fig. 3 includes a `for`-loop $\mathtt{F}$. This loop, on input $\langle \mathcal{I}_0, \mathcal{I}_1 \rangle$, sums in $\mathtt{Z}$ the squares of the numbers which, in the Fibonacci sequence, have indexes from $\mathcal{I}_0$ to $\mathcal{I}_0 + \mathcal{I}_1 - 1$: if e.g. $\mathcal{I}_0 = 4$ and $\mathcal{I}_1 = 3$, then $\mathtt{Z}$ finally evaluates to $3^2 + 5^2 + 8^2 = 98$. Whenever a program $\mathtt{P}$ includes a `for`-loop $\mathtt{F}$, we write $\mathtt{F} \in \mathsf{fors}(\mathtt{P})$. More formally, $\mathtt{F} \in \mathsf{fors}(\mathtt{P})$ iff $\mathtt{F} \subseteq \mathtt{P}$ and $\mathtt{F} \triangleq \{\mathtt{G}, \bar{\mathtt{G}}, \mathtt{I}\} \cup \mathtt{H}$. The couple of commands $\mathtt{G} \triangleq \mathtt{g}\colon \mathtt{X} < \dot{\mathtt{E}} \to \mathtt{h};$ and $\bar{\mathtt{G}} \triangleq \mathtt{g}\colon \neg(\mathtt{X} < \dot{\mathtt{E}}) \to \mathtt{p};$, with $\mathtt{g} \neq \mathtt{h}$ and $\mathtt{g} \neq \mathtt{p}$, implements a branching named *guard*. As $\mathtt{F}$ always starts with the evaluation of its guard, we have $\mathfrak{L}(\mathtt{F}) = \{\mathtt{g}\}$, $\mathsf{lab}(\mathtt{F}) \cap \mathsf{lab}(\mathtt{P} \setminus \mathtt{F}) = \emptyset$ and $\mathsf{suc}(\mathtt{P} \setminus \mathtt{F}) \cap \mathsf{lab}(\mathtt{F}) \subseteq \{\mathtt{g}\}$. The guard is satisfied as long as $\mathtt{X} \in \mathbb{X}$ is less[2] than $\dot{\mathtt{E}} \in \mathbb{E}$. If the guard is not satisfied, the `for`-loop ends transferring the control flow at entrypoint $\mathtt{p} \notin \mathsf{lab}(\mathtt{F})$. Otherwise, the execution goes on through $\mathtt{H}$, a set of commands named *body*, and eventually through an *increment* command $\mathtt{I} \triangleq \mathtt{i}\colon \mathtt{X} := \mathtt{X} + \ddot{\mathtt{E}} \to \mathtt{g};$, with $\mathtt{i} \neq \mathtt{g}$ and $\mathtt{i} = \mathtt{h} \vee \mathtt{i} \in \mathsf{suc}(\mathtt{H})$; notice that $\mathtt{I}$ makes the control flow return to the guard again. We formally define $\mathtt{H}$ as the collection of all the commands of $\mathtt{P}$ that are reachable from $\mathtt{G}$ without going through $\mathtt{I}$, i.e., $\mathtt{H} \triangleq \mathsf{lfp}^{\subseteq} \mathsf{flow}(\mathtt{P})$, where $\mathsf{flow}(\mathtt{P})(\mathtt{Q}) \triangleq \{\mathtt{C} \in \mathtt{P} \setminus \{\mathtt{I}\} \mid \mathsf{lab}(\mathtt{C}) = \mathsf{suc}(\mathtt{G}) \vee \exists \mathtt{C}' \in \mathtt{Q}.\ \mathsf{lab}(\mathtt{C}) = \mathsf{suc}(\mathtt{C}')\}$. We require $\mathtt{g}, \mathtt{i} \notin \mathsf{lab}(\mathtt{H})$. We expect both $\mathtt{X}$ and the variables in $\dot{\mathtt{E}}$ and $\ddot{\mathtt{E}}$ not to be assigned inside $\mathtt{H}$. We require $\mathtt{X}$ not to be used in $\dot{\mathtt{E}}$ or $\ddot{\mathtt{E}}$.

Finite partial trace $\langle \rho, \mathtt{G} \rangle \eta \langle \rho', \mathtt{I} \rangle \in \mathsf{S}(\mathtt{F})$ is an *iteration of* `for`*-loop* $\mathtt{F}$, where $\eta \in \mathsf{S}(\mathtt{H})$; if $\mathtt{H} = \emptyset$ then $\eta = \varepsilon$. A maximal trace $\sigma \in \mathsf{S}(\mathtt{F})$ is a sequence of terminating iterations[3] followed by a state with command $\bar{\mathtt{G}}$. Along the trace, the values of $\dot{\mathtt{E}}$ and $\ddot{\mathtt{E}}$ do not change, while the value of $\mathtt{X}$, though constant throughout each iteration, increases by $\ddot{\mathtt{E}}$ from one iteration to another. Thus, if ρ is an environment in a state of $\sigma \in \mathsf{S}(\mathtt{F})$, we can predict how many increments $\mathtt{X}$ still has to undergo, i.e., the number of the iterations from ρ till the end of σ. Let $\dot{e} \triangleq \mathsf{A}(\dot{\mathtt{E}})\rho$, $\ddot{e} \triangleq \mathsf{A}(\ddot{\mathtt{E}})\rho$ and $x \triangleq \mathsf{A}(\mathtt{X})\rho$. We just need to define $\alpha_{\mathtt{F}} : \mathfrak{E}(\mathtt{F}) \to \mathbb{N}$ such that $\alpha_{\mathtt{F}}(\rho) \triangleq \left\lfloor \frac{(\dot{e}-x)+(\ddot{e}-1)}{\ddot{e}} \right\rfloor$ if $\dot{e} \geq x$ and $\alpha_{\mathtt{F}}(\rho) \triangleq 0$ otherwise. We let ι be the total number of iterations of σ.

Along $\sigma \in \mathsf{S}(\mathtt{F})$ iterations are naturally unfolded, i.e., they come sequentially one after another. In $\mathbb{p}(\{\sigma\})$ they fold because any command $\mathtt{C} \in \mathtt{F}$, although occurring in many different iterations, always appears with the same entrypoint $\mathsf{lab}(\mathtt{C})$. In the proposed watermarking technique, folding has to be neutralized at embedding/extraction time. Loop-unrolling [2] is good at this task because it changes labels in the following way: given the so-called *unrolling factor* $u \in \mathbb{N}$, it makes all and only the occurrences of $\mathtt{C}$ at iterations $k \pmod{u}$ have the same label (with $0 \leq k < \iota$), thus partitioning the iterations of σ into u classes. Only iterations from the same class fold together. So the code of the unrolled loop is u times longer than $\mathtt{F}$ and each of its iterations sequentially executes the task of u native iterations. Consider for instance `for`-loop $\mathtt{F}' \in \mathsf{fors}(\mathtt{P}')$ in Fig. 3, which has a command $\bar{\mathtt{C}}$ with entrypoint $00\mathtt{h}$. Clearly $\bar{\mathtt{C}}$ appears in every iteration of any $\sigma \in \mathsf{S}(\mathtt{F}')$. Now let $\sigma = \sigma'\sigma''$, where σ' encompasses the first $0 \leq \iota_1 \leq \iota$ iterations and σ'' the last $\iota_2 = \iota - \iota_1$ ones. To unroll σ'' with factor $u_2 = 3$, we scan

[2] For short, we ignore similar kinds of `for`-loops, which use $>$, $\leq$ or $\geq$ as comparison operator.

[3] An iteration also might not conclude: this occurs when the execution of $\mathtt{F}$ gets trapped inside some non-terminating loops possibly included in $\mathtt{H}$. In such a case none of the partial traces of $\mathsf{S}(\mathtt{F})$ can be recognized as a maximal trace which fully outlines the entire execution.

$$\begin{aligned}
\mathtt{Y} := ?[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}] &\triangleq \mathtt{Y} := ?\\
\mathtt{Y} := \mathtt{E}[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}] &\triangleq \mathtt{Y} := \mathtt{E}[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}]\\
\mathtt{B}_1 \mathbin{@} \mathtt{B}_2[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}] &\triangleq \mathtt{B}_1[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}] \mathbin{@} \mathtt{B}_2[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}]\\
\neg\mathtt{B}[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}] &\triangleq \neg\mathtt{B}[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}]\\
{}^{\mathtt{tt}}/_{\mathtt{ff}}[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}] &\triangleq {}^{\mathtt{tt}}/_{\mathtt{ff}}\\
\mathtt{E}_1 \mathbin{@} \mathtt{E}_2[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}] &\triangleq \mathtt{E}_1[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}] \mathbin{@} \mathtt{E}_2[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}]\\
\mathtt{n}[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}] &\triangleq \mathtt{n}\\
\mathtt{Y}[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}] &\triangleq \mathtt{Y} \quad (\mathtt{Y} \neq \mathtt{X})\\
\mathtt{X}[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}] &\triangleq \mathtt{X} + m \times \ddot{\mathtt{E}} \quad (m \neq 0)\\
\mathtt{X}[{}^{(\mathtt{X}+0\times\ddot{\mathtt{E}})}/_{\mathtt{X}}] &\triangleq \mathtt{X}
\end{aligned}$$

$$\mathsf{I}(i,\mathtt{C}) \triangleq \begin{cases} i+1 & \text{if } i < I \wedge (\mathtt{C} = \bar{\mathtt{G}} \\ & \quad \vee (\mathtt{C} \notin \mathtt{F} \wedge \mathsf{suc}(\mathtt{C}) = 00\mathrm{g})) \\ 0 & \text{if } i = I \wedge \mathtt{C} = \bar{\mathtt{G}} \\ i & \text{otherwise} \end{cases}$$

$$\mathsf{M}(m,i,\mathtt{I}) \triangleq \begin{cases} m+1 & \text{if } m \in [0, u_i - 1) \\ 0 & \text{if } m = u_i - 1 \end{cases}$$

$$\mathsf{M}(m,i,\mathtt{C}) \triangleq m \qquad \text{if } \mathtt{C} \neq \mathtt{I}$$

$$\begin{aligned}
&t^{\mathrm{lu}}(\varepsilon\langle\rho,\mathtt{C}\rangle) \triangleq \textbf{let } i = \textbf{if } \mathtt{C} \in \mathtt{F} \textbf{ then } 1 \textbf{ else } 0 \textbf{ in } t^{\mathrm{lu}}(\langle\rho,\mathtt{C}\rangle, 0, i) \textbf{ fi}\\
&t^{\mathrm{lu}}(\sigma'\langle\rho',\mathtt{C}'\rangle\langle\rho,\mathtt{C}\rangle) \triangleq \textbf{let } \overline{\sigma}\langle\overline{\rho}, \mathtt{L}\colon \mathtt{A} \rightarrow ims\mathtt{;}\rangle = t^{\mathrm{lu}}(\sigma'\langle\rho',\mathtt{C}'\rangle) \textbf{ in } \overline{\sigma}\langle\overline{\rho}, \mathtt{L}\colon \mathtt{A} \rightarrow ims\mathtt{;}\rangle t^{\mathrm{lu}}(\langle\rho,\mathtt{C}\rangle, m, i)\\
&t^{\mathrm{lu}}(\langle\rho,\mathtt{C}\rangle, m, i) \triangleq \mathsf{V}(\rho[\mathtt{X} := \mathsf{A}(\mathtt{X})\rho - m\mathsf{A}(\ddot{\mathtt{E}})\rho], t^{\mathrm{lu}}(\mathtt{C}, m, i))
\end{aligned}$$

$$\begin{aligned}
&t^{\mathrm{lu}}(00\mathtt{s}\colon \mathtt{A} \rightarrow 00\mathtt{s}'\mathtt{;}, m, i) \triangleq \textbf{let } \langle m', i'\rangle = \langle \mathsf{M}(m, i, 00\mathtt{s}\colon \mathtt{A} \rightarrow 00\mathtt{s}'\mathtt{;}), \mathsf{I}(i, 00\mathtt{s}\colon \mathtt{A} \rightarrow 00\mathtt{s}'\mathtt{;})\rangle \textbf{ in}\\
&\textbf{let } \langle \mathtt{L}, \mathtt{L}'\rangle = \langle ims, i'm'\mathtt{s}'\rangle \textbf{ in let } \mathtt{B} = \mathtt{X} < \dot{\mathtt{E}} - (\ell_i + u_i - 1) \times \ddot{\mathtt{E}} \textbf{ in match } \mathtt{C} \textbf{ with}\\
&\bar{\mathtt{G}} \quad \longmapsto \textbf{if } m > 0 \textbf{ then } \mathtt{L}\colon \mathtt{ff} \rightarrow \mathtt{L}\mathtt{;} \textbf{ else if } i = 0 \textbf{ then } \mathtt{L}\colon \neg\mathtt{B} \rightarrow \mathtt{L}'\mathtt{;}\\
&\qquad\qquad \textbf{else } \mathtt{L}\colon \neg\mathtt{B} \rightarrow i'm'\mathtt{g}\mathtt{;}\ t^{\mathrm{lu}}(\bar{\mathtt{G}}, m', i') \textbf{ fi}\\
&\mathtt{G} \quad \longmapsto \textbf{if } m > 0 \textbf{ then } \mathtt{L}\colon \mathtt{tt} \rightarrow \mathtt{L}'\mathtt{;} \textbf{ else if } i = 0 \textbf{ then } \mathtt{L}\colon \mathtt{B} \rightarrow \mathtt{L}'\mathtt{;}\\
&\qquad\qquad \textbf{else let } i'' = \mathsf{I}(i, \bar{\mathtt{G}}) \textbf{ in } \mathtt{L}\colon \mathtt{B} \rightarrow \mathtt{L}'\mathtt{;}\ \mathtt{L}\colon \neg\mathtt{B} \rightarrow i''m'\mathtt{g}\mathtt{;}\ t^{\mathrm{lu}}(\mathtt{G}, m', i'') \textbf{ fi}\\
&\mathtt{I} \quad \longmapsto \textbf{if } m = u_i - 1 \textbf{ then } \mathtt{L}\colon \mathsf{act}(\mathtt{C})[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}] \rightarrow \mathtt{L}'\mathtt{;} \textbf{ else } \mathtt{L}\colon \mathtt{tt} \rightarrow \mathtt{L}'\mathtt{;} \textbf{ fi}\\
&\_ \quad \longmapsto \mathtt{L}\colon \mathsf{act}(\mathtt{C})[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}] \rightarrow \mathtt{L}'\mathtt{;}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{V}(\rho, \mathtt{ls}) \triangleq \textbf{match } \mathtt{ls} \textbf{ with}\\
&\mathtt{L}\colon \mathtt{B} \rightarrow \mathtt{L}'\mathtt{;}\ \mathtt{L}'\colon \neg\mathtt{B} \rightarrow \mathtt{L}''\mathtt{;}\ \mathtt{ls}' \longmapsto \textbf{if } \mathsf{B}(\mathtt{B})\rho \textbf{ then } \langle\rho, \mathtt{L}\colon \mathtt{B} \rightarrow \mathtt{L}'\mathtt{;}\rangle \textbf{ else } \langle\rho, \mathtt{L}'\colon \neg\mathtt{B} \rightarrow \mathtt{L}''\mathtt{;}\rangle\mathsf{V}(\rho, \mathtt{ls}') \textbf{ fi}\\
&\mathtt{C}\ \mathtt{ls}' \longmapsto \langle\rho, \mathtt{C}\rangle\mathsf{V}(\rho, \mathtt{ls}')\\
&\varepsilon \longmapsto \varepsilon
\end{aligned}$$

Fig. 5. Semantic loop-unrolling

the iterations of σ'' by triplets; for each triplet, we set the memory value of $\mathsf{lab}(\overline{\mathtt{C}})$ to 0 in the first iteration, 1 in the second iteration and to 2 in the third iteration. If we fold the new trace, we obtain `for`-loop $\mathtt{F}_2$ of program $\mathtt{Q}$ in Fig. 3: here three copies of $\overline{\mathtt{C}}$ coexists at entrypoints $20\mathtt{h}$, $21\mathtt{h}$ and $22\mathtt{h}$. Similarly, by unrolling σ' with the trivial factor $u_1 = 1$, we get `for`-loop $\mathtt{F}_1$ of $\mathtt{Q}$, in which the only one copy of $\overline{\mathtt{C}}$ is located at $10\mathtt{h}$. All the copies of $\overline{\mathtt{C}}$ have the same symbol $\mathtt{h}$. As index value, they use a number identifying the unrolled loop which they are member of. The fact that the code of the unrolled loops actually implements tuples of native iterations is essential to the proposed watermarking technique. We hide signatures in iterations, which are semantic objects. However, the embedder and the extractor are automatic tools that cannot deal with semantics. But they can deal with code. Thus if we define loop-unrolling as a semantic transformation and then we abstract it to a syntactic transformation [12], we can safely rely on loop-unrolling to both embed and extract signatures. In our last example we unrolled $\sigma = \sigma'\sigma''$ using u_1 only for σ'. To attain this, we kept on unrolling σ only while $\mathtt{X} < \dot{\mathtt{E}} - (\ell_1 + u_1 - 1) \times \ddot{\mathtt{E}}$ was true, where we let $\ell_1 = \iota_2$. This approach was supported by the following proposition. Define $\ell \in [0, \iota]$ to be the *lessening factor*. Let $g(u,\ell) \triangleq \ell + u - 1$ and $\hat{\mathtt{B}} \triangleq \mathtt{X} < \dot{\mathtt{E}} - g(u,\ell) \times \ddot{\mathtt{E}}$. Let $\rho \in \mathfrak{E}(\mathtt{F})$ be an environment in σ.

Proposition 2. $\mathsf{B}(\hat{\mathtt{B}})\rho = \mathsf{ff}$ *if and only if* $0 \leq \alpha_{\mathtt{F}}(\rho) \leq g(u,\ell)$*, i.e.,* $\hat{\mathtt{B}}$ *gets false in* σ *at the last but* $g(u,\ell)$ *iteration. Moreover* $\lfloor{}^{(\iota-\ell)}/_{u}\rfloor\, u - u < \iota - g(u,\ell) \leq \lfloor{}^{(\iota-\ell)}/_{u}\rfloor\, u$.

So the unrolling of σ with u_1, ℓ_1 involved just the first $\lfloor{}^{(\iota-\ell_1)}/_{u_1}\rfloor\, u_1$ iterations. This did not keep us from unrolling unprocessed iterations with new factors $u_2 = 3$ and $\ell_2 = 0$.

As we know, loop-unrolling affects labels. Consider again `for`-loop $\mathtt{F}_2$ in Fig. 3: in iterations $k \pmod{u_2}$ each 00s was replaced with $2m$s, where $m \triangleq k \bmod u_2$ is the memory value. But loop-unrolling affects actions as well: each iteration of $\mathtt{F}_2$, for instance, stemmed from the merger of $u_2 = 3$ subsequent native iterations. The process was as follows. Guards and increments were replaced by tt in every iteration of σ'', except for iterations $0 \pmod{u_2}$, where $\hat{\mathtt{B}}_2 = \mathcal{I}_0 - 4$ was used as the new guard, and for iterations $(u_2 - 1) \pmod{u_2}$, where $\mathsf{act}(\mathtt{I})[{}^{(\mathtt{X}+(u_2-1)\times\ddot{\mathtt{E}})}/_{\mathtt{X}}]$ – see Fig. 5 – was used as the new increment. In iterations $k \pmod{u_2}$, any other $\mathsf{act}(\mathtt{C})$ was replaced with $\mathsf{act}(\mathtt{C})[{}^{(\mathtt{X}+m\times\ddot{\mathtt{E}})}/_{\mathtt{X}}]$, and every environment ρ was updated to $\rho[\mathtt{X} := \mathsf{A}(\mathtt{X})\rho - m\mathsf{A}(\ddot{\mathtt{E}})\rho]$. After $\lfloor{}^{(\iota-\ell_2)}/_{u_2}\rfloor\, u_2$ iterations of σ, $\hat{\mathtt{B}}_2$ becomes false; thus here a new state with command 20g: $\neg\hat{\mathtt{B}}_2 \to$ 00g; was inserted. Such new states are discarded by observational abstraction $\alpha^{\mathrm{lu}}_{\mathcal{O}}$ for loop-unrolling which, for any other state $\langle\rho, \mathtt{C}\rangle$, gets rid of C and reverses the update of ρ using the memory value inside $\mathsf{lab}(\mathtt{C})$:

$$\alpha^{\mathrm{lu}}_{\mathcal{O}}(\mathcal{T}) \triangleq \{\alpha^{\mathrm{lu}}_{\mathcal{O}}(\sigma) \mid \sigma \in \mathcal{T}\} \qquad \alpha^{\mathrm{lu}}_{\mathcal{O}}(\sigma) \triangleq \lambda j.\ \alpha^{\mathrm{lu}}_{\mathcal{O}}(\sigma_j)$$
$$\alpha^{\mathrm{lu}}_{\mathcal{O}}(\langle\rho, im\mathtt{s}\texttt{:}\ \mathtt{A} \to im\mathtt{s}'\texttt{;}\rangle) \triangleq \textbf{if}\ (\mathtt{s} = \mathtt{s}')\ \varepsilon\ \textbf{else}\ \rho[\mathtt{X} := \mathsf{A}(\mathtt{X})\rho + m\mathsf{A}(\ddot{\mathtt{E}})\rho]\ \textbf{fi}\ .$$

Semantic transformation t^{lu} for loop-unrolling, shown in Fig. 5, scans a trace in $\mathsf{S}(\mathtt{P})$ and unrolls any subtrace $\sigma \in \mathsf{S}(\mathtt{F})$, using factors from vectors $\mathbf{u} = \langle u_1, \ldots, u_I\rangle$ and $\boldsymbol{\ell} = \langle \ell_1, \ldots, \ell_I\rangle$. For each $u_i \geq 1, \ell_i \geq 0$ it produces unrolled `for`-loop $\mathtt{F}_i$. Native iterations left unprocessed in the rear of σ belong to $\mathtt{F} = \mathtt{F}_0$, where the equality holds since $u_0 \triangleq 1$ and $\ell_0 \triangleq 0$. Index i, initially set to 0, is ruled by function I, which increases it just at the beginning of σ and after the insertion of each new state. When the unrolling is over, I reverts i to 0. While unrolling is performed ($i > 0$), $\hat{\mathtt{B}}_i$ has to be checked and inserted every u_i iterations of σ. The count is kept through memory value m controlled by function M. The check is performed by validation function V, and it occurs whenever $m = 0$ and t^{lu} is about to transform a guard state. In particular, if $\hat{\mathtt{B}}_i$ evaluates to false, the additional state is inserted and then unrolling goes on using the next factors, if any, provided that there are still native iterations to unroll.

Proposition 3. *Let* P *be a program and* $\sigma \in \mathsf{S}(\mathtt{P})$*. Then* $t^{\mathrm{lu}}(\sigma)$ *is a trace and* $\alpha^{\mathrm{lu}}_{\mathcal{O}}(t^{\mathrm{lu}}(\sigma)) = \alpha^{\mathrm{lu}}_{\mathcal{O}}(\sigma)$*. Furthermore* $\mathbb{p} \circ (t^{\mathrm{lu}} \circ \mathsf{F}) = \mathbb{F}^{\mathrm{lu}} \circ \mathbb{p}$.

t^{lu} turns a trace $\sigma \in \mathsf{S}(\mathtt{F})$ into an $\alpha^{\mathrm{lu}}_{\mathcal{O}}$-equivalent trace $\sigma' \in \mathsf{S}(\mathtt{F}_1 \cup \ldots \cup \mathtt{F}_I \cup \mathtt{F})$, notwithstanding σ is a subtrace inside a trace of P. Thanks to the fixpoint transfer, we get the algorithm yielding $\mathtt{P}' \triangleq \mathtt{P} \cup \mathtt{F}_1 \cup \ldots \cup \mathtt{F}_I$ from $\mathtt{P} \supseteq \mathtt{F}$. We express it as follows:

$$\textsc{Init}^{\text{lu}}(\mathtt{P}) \triangleq \{\mathtt{C}' \mid \exists \mathtt{C} \in \mathtt{P}.\, \mathsf{lab}(\mathtt{C}) \in \mathfrak{L}(\mathtt{P}) \ \wedge \mathtt{C}' \text{ is in } t^{\text{lu}}(\mathtt{C}, 0, \textbf{if } \mathtt{C} \in \mathtt{F} \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi})\}$$

$$\textsc{Next}^{\text{lu}}(\mathtt{P})(\mathtt{Q}) \triangleq \{\mathtt{C}' \mid \exists \mathtt{C} \in \mathtt{P}.\, \exists \mathtt{L}\mathtt{:}\ \mathtt{A} \to i'm'\mathtt{s}'\mathtt{;} \in \mathtt{Q}.\, \mathsf{lab}(\mathtt{C}) = 00\mathtt{s}' \wedge \mathtt{C}' \text{ is in } t^{\text{lu}}(\mathtt{C}, m', i')\}$$

$$\textsc{Iter}^{\text{lu}}(\mathtt{P})(\mathtt{Q}) \triangleq \textbf{ let } \mathtt{Q}' = \mathtt{Q} \cup \textsc{Next}^{\text{lu}}(\mathtt{P})(\mathtt{Q}) \textbf{ in if } \mathtt{Q}' = \mathtt{Q} \ \textbf{ then } \mathtt{Q}' \textbf{ else } \textsc{Iter}^{\text{lu}}(\mathtt{P})(\mathtt{Q}') \textbf{ fi} \ .$$

5 Software Watermarking by Loop-Unrolling

In the watermarking technique we propose here, a signature is a natural number $\mathfrak{s}$, which is computed iteratively in watermark variable W by mean of the following stegomark: $\mathtt{W} := a;$ **for** $\mathtt{X} := \mathtt{0}$ **to** $n-1$ **do** $\mathtt{W} := \xi \times \mathtt{W} + b;$ **od**. This stegomark implements the Horner technique for the evaluation at $x = \xi$ of n-degree polynomial $P_n(x) \triangleq ax^n + b\sum_{j=0}^{n-1} x^j$. Hence we have $\mathfrak{s} = P_n(\xi)$. Let us consider an example: signature $\mathfrak{s} = 120736$, obtained as the evaluation at $x = 14$ of the 3-degree polynomial $P_3(x) = -199948x^3 + 245760x^2 + 245760x + 245760$, can be computed by the following stegomark: $\mathtt{W} := -\mathtt{199948};$ **for** $\mathtt{X} := \mathtt{0}$ **to** 2 **do** $\mathtt{W} := \mathtt{14} \times \mathtt{W} + \mathtt{245760};$ **od**. The degree of P_n is precisely the number n of iterations performed by the `for`-loop in the stegomark. The stegomark is going to be embedded in a `for`-loop $\mathtt{F} \in \mathsf{fors}(\mathtt{P})$ performing, on some input $\mathcal{I}$, at least n iterations. Thus n can range from 1 to the maximum number of iterations F can perform when P is executed. Reasonably we assume that for any `for`-loop $\mathtt{F} \in \mathsf{fors}(\mathtt{P})$ there exists an input $\mathcal{I}$ such that F performs at least one iteration on $\mathcal{I}$. Thus any `for`-loop can be targeted for embedding. Likely, we expect that in any program which is complex enough to be worth protection there is at least a `for`-loop where to embed the stegomark. This because, in such programs, large amounts of data aggregate in data structures, like e.g. arrays, that need `for`-loops to be manipulated.

Given $\mathfrak{s}$ and $n > 0$, we would like $P_n(\xi) = a\xi^n + b\sum_{j=0}^{n-1} \xi^j = \mathfrak{s}$. We thereby let $a \triangleq \frac{\mathfrak{s}}{\xi^n} - \frac{b}{\xi^n}\sum_{j=0}^{n-1} \xi^j$. We ask for ξ, b and a to be whole numbers, so that $\mathfrak{s}$ can be safely evaluated through the stegomark. First, we require ξ^n to be a divisor of $\mathfrak{s}$. In our example we have $\mathfrak{s} = 120736 = 2^5 \cdot 7^3 \cdot 11$ and $n = 3$, so $\xi = 14$ is one possible choice. Next, we require b to be a nonzero multiple of ξ^n, namely $b \neq 0$ and $b = \xi^{n+n'}z$, where $n' \in \mathbb{N}$ and $z \in \mathbb{Z}$ are random numbers. In our example we set $b = 14^{3+11} \cdot 15 = 245760$. As watermarked program Q in Fig. 3 shows, ξ and b are not obfuscated. Moreover, by design, it is known that b is a multiple of ξ^n. If n' was fixed by design, e.g. $n' \triangleq 0$, then n could be easily retrieved – by just subtracting n' to the number q of times ξ divides b. This would be unpleasant because n is part of the secret watermarking key. By letting n' be selected randomly, what it is known to an attacker is that $0 < n \leq q$: the greater is n', the larger is the range of n. Programming languages do not allow numbers to exceed a prefixed maximum *MAX*. If parameters ξ and b are too big, we may compute them using ad-hoc functions fed with smaller values; this also increases the stealth of such parameters.

Embedding. In order to inlay the stegomark computing $\mathfrak{s}$ in P, we run the embedding algorithm shown in Fig. 6. The algorithm looks for a `for`-loop F that, on a given input $\mathcal{I}$, performs at least n iterations. If the guard of F includes variables that are initialized randomly, the number of iterations on $\mathcal{I}$ may not be fixed. Therefore we let ι be the minimum number of iterations of F on $\mathcal{I}$, and we require $\iota \geq n$. Furthermore, stegomark

```
**funct** $\textit{embed}(\mathtt{P},\mathcal{I},\mathtt{W},n,\xi,a,b)$
$\mathtt{P}' \leftarrow \mathtt{P}$;  $\mathcal{F} \leftarrow \mathsf{fors}(\mathtt{P})$;
**while** $\mathcal{F} \neq \emptyset \wedge \mathtt{P}' = \mathtt{P}$ **do**
$\mathtt{F} \leftarrow \textit{next}(\mathcal{F})$;     (by def. $\mathtt{F} \triangleq \{\bar{\mathtt{G}},\mathtt{G},\mathtt{I}\} \cup \mathtt{H}$)
$\mathcal{F} \leftarrow \mathcal{F} \setminus \{\mathtt{F}\}$;
$\iota \leftarrow$ min # iterations of $\mathtt{F}$ when $\mathtt{P}$ is run on $\mathcal{I}$;
**if** ($\iota \geq n \wedge \mathtt{W}$ is dead wrt. the guard of $\mathtt{F}$)
  $\mathbf{u} \leftarrow \langle\iota\rangle$;  $\boldsymbol{\ell} \leftarrow \langle 0\rangle$;
  $\mathtt{Q} \leftarrow \mathfrak{t}^{\mathrm{lu}}(\mathtt{P};\ \mathtt{F},\mathbf{u},\boldsymbol{\ell})$;
  **if** (there exists $\mathtt{C} \in \mathtt{Q}$ such that
      $\mathtt{C} = \mathtt{L}\texttt{:}\ \mathtt{A} \rightarrow \mathtt{L}'\texttt{;}$
      $\mathtt{L} = 1\delta\mathtt{s}$     with $\mathtt{s} \neq \mathtt{i} \wedge \delta \in [0,\iota-n]$
      $\mathtt{A} = \mathtt{Y} := \mathtt{E}$ with $\mathtt{Y} \in \mathsf{var}(\mathtt{Q}) \wedge \mathtt{E} \in \mathbb{E}$
      $\mathtt{L}' = 1\delta\mathtt{v}$     with $\mathtt{v} \in \mathsf{lab}(\mathtt{F})$)
    $\mathtt{S} \leftarrow \textit{slice}(\mathtt{Q},\mathtt{C})$;
    $y \leftarrow$ value of $\mathtt{Y}$ when $\mathtt{S}$ is run on $\mathcal{I}$;
    $r_0 \leftarrow$ a random number in $\mathbb{Z} \setminus \{y\}$;
    $r_1 \leftarrow$ a random number in $\mathbb{Z}$;
    let $f(\mathtt{Y}) \triangleq \dfrac{r_1 - a}{r_0 - y}(\mathtt{Y} - y) + a$;
    $\mathtt{w} \leftarrow$ a label from $\mathsf{lab}(\mathtt{H} \cup \{\mathtt{I}\})$ that
        is reached after $\mathtt{v}$ in the CFG of $\mathtt{P}$;
    $\theta_0 \leftarrow \langle \mathtt{v},\mathtt{W},f(\mathtt{Y})\rangle$;
    $\theta_1 \leftarrow \langle \mathtt{w},\mathtt{W},\xi \times \mathtt{W} + b\rangle$;
    $\mathtt{P}' \leftarrow \mathfrak{t}^{\mathrm{in}}(\mathtt{P};\ \theta_0,\theta_1)$;  **fi fi od**
**return** $\langle \mathtt{P}',\langle \mathcal{I},\delta,n\rangle\rangle$;
```

```
**funct** $\textit{extract}(\mathtt{P}',\langle \mathcal{I},\delta,n\rangle)$
$\mathcal{S} \leftarrow \emptyset$;
**for each** $\mathtt{F} \in \mathsf{fors}(\mathtt{P}')$ **do**
  $\iota \leftarrow$ # iterations of $\mathtt{F}$ when $\mathtt{P}'$ is run on $\mathcal{I}$;
  **if** ($\iota \geq n$)
    $\mathbf{u} \leftarrow \langle 1,n\rangle$;
    $\boldsymbol{\ell} \leftarrow \langle \iota-\delta,\iota-\delta-n\rangle$;
    $\mathtt{Q} \leftarrow \mathfrak{t}^{\mathrm{lu}}(\mathtt{P}';\ \mathtt{F},\mathbf{u},\boldsymbol{\ell})$;
    **for each** $\mathtt{L}\texttt{:}\ \mathtt{A} \rightarrow \mathtt{L}'\texttt{;} \in \mathtt{Q}$ such that
      $\mathtt{L} = 20\mathtt{s}$     with $\mathtt{s} \in \mathbb{S}$
      $\mathtt{A} = \mathtt{W} := \mathtt{E}$ with $\mathtt{W} \in \mathsf{var}(\mathtt{Q}) \setminus \mathsf{var}(\mathtt{E})$
                      $\wedge\, \mathtt{E} \in \mathbb{E}$
      $\mathtt{L}' = 20\mathtt{s}'$   with $\mathtt{s} \in \mathbb{S}$
    **do**
    $\mathtt{Q}' \leftarrow \mathtt{Q} \setminus \{\mathtt{C} \in \mathtt{Q} \mid$
            $\exists m > 0.\ \mathsf{lab}(\mathtt{C}) = 2m\mathtt{s}\}$;
    $\mathtt{R} \leftarrow \{\mathtt{L}''\texttt{:}\ \mathtt{A}' \rightarrow \mathtt{L}'''\texttt{;} \in \mathtt{Q}' \mid$
        $\mathtt{L}'' = 2[n-1]\mathtt{s}''$ with $\mathtt{s}'' \in \mathbb{S}$
        $\mathtt{A}' = \mathtt{W} := \mathtt{E}'$ with $\mathtt{W} \in \mathsf{var}(\mathtt{E})$
                          $\wedge'\, \mathtt{E} \in \mathbb{E}$
        $\mathtt{L}''' = 2[n-1]\mathtt{s}'''$ with $\mathtt{s}''' \in \mathbb{S}\}$
    **if** ($\mathtt{R}$ is a singleton with element $\mathtt{C}$)
      $\mathtt{S} \leftarrow \textit{slice}(\mathtt{Q}',\mathtt{C})$;
      $w \leftarrow$ value of $\mathtt{W}$ when $\mathtt{S}$ is run on $\mathcal{I}$;
      $\mathcal{S} \leftarrow \mathcal{S} \cup \{w\}$;  **fi od fi od**
**return** $\mathcal{S}$;
```

Fig. 6. Embedding and extraction algorithms

variable $\mathtt{W}$ must be dead during the execution of $\mathtt{F}$. If such a `for`-loop does not exists in $\mathtt{P}$, the algorithm fails and returns $\mathtt{P}$ and the empty key. Otherwise, it gets from $\mathtt{F}$ an unrolled `for`-loop $\mathtt{F}_1$ which syntactically displays all the ι iterations as sequential code: actually, any command $\mathtt{C}' \in \mathtt{F}_1$ derived from iteration $m \in [0,\iota)$ is such that $\exists \mathtt{s} \in \mathbb{S}.\ \mathsf{lab}(\mathtt{C}') = 1m\mathtt{s}$; here we also say that $\mathtt{C}'$ is at *offset* m. Next, the algorithm looks for a command $\mathtt{C}$ at offset $\delta \in [0,\iota-n)$ such that $\mathsf{act}(\mathtt{C}) = \mathtt{Y} := \mathtt{E}$. If it succeeds, it computes actual value y of $\mathtt{Y}$ on input $\mathcal{I}$, using backward-slicing with criterion $\mathtt{C}$. Then it lets first-degree polynomial $f(\mathtt{Y})$ to model the line passing through points (y,a) and (r_0,r_1) in the Cartesian coordinate system; in such a way, one possible dependence between y and parameter a of the stegomark is established. Finally, the algorithm comes back to subject program $\mathtt{P}$, and it inserts $\mathtt{W} := f(\mathtt{Y})$ at entrypoint $\mathsf{lab}(\mathtt{C})$ and $\mathtt{W} := \xi \times \mathtt{W} + b$ somewhere below, inside the body of $\mathtt{F}$. In such a way, it obtains marked program $\mathtt{P}'$ which it returns together with key $\langle \mathcal{I},\delta,n\rangle$. Note that we can guarantee $f(\mathtt{Y}) = a$ only at offset δ. If $\mathtt{Y}$ denotes stochastic behavior, i.e., it changes its own value from one iteration to another, the knowledge of δ becomes essential at extraction time to get the correct initialization of $\mathtt{W}$. This improves reliability and stealth of the watermark. The iteration at offset δ is the *promoter* of the signature recovery, and δ measures its displacement in the sequence of the ι iterations.

As shown in Fig. 3, the embedding phase basically consists in a pair of assignment insertions. To inlay in $\mathtt{P}$ our signature $\mathfrak{s} = P_3(14) = 120736$, we want the `for`-loop to perform at least $n = 3$ iterations, so we let $\mathcal{I}_0 \triangleq 8$, obtaining $\iota = 8$. Furthermore, by fixing $\mathcal{I}_1 \triangleq 13$ and $f \triangleq \lambda\mathtt{Y}.\ (\mathtt{215} - \mathtt{Y}) \times \mathtt{259}$, we ensure that, at entry point $\mathtt{v}$ of the iteration at offset $\delta = 3$, we have $y = \mathsf{Fib}(\mathtt{X} + \mathcal{I}_1) = \mathsf{Fib}(3 + 13) = 987$ and $f(987) = -199948 = a$. Thus, once we have chosen label $\mathtt{w} \equiv \mathtt{i}$ as target entry point for the second assignment, we insert $\mathtt{W} := f(\mathtt{Y})$ at $\mathtt{v}$ and $\mathtt{W} := \mathtt{14} \times \mathtt{W} + \mathtt{245760}$ at $\mathtt{w}$.

Extraction. To extract our signature $\mathfrak{s}$ from marked program $\mathtt{P}'$, we need to deliver $\mathtt{P}'$ and key $\kappa = \langle \mathcal{I}, \delta, n \rangle$ to the algorithm described in Fig. 6. From each `for`-loop $\mathtt{F}$ in $\mathtt{P}'$ performing on input $\mathcal{I}$ a number $\iota \geq n$ of iterations, the algorithm tries to gain a set of candidate signatures; the final result of the extraction is a union set $\mathcal{S}$ collecting altogether the candidate signatures coming from each set. To gain a set of candidate signatures from $\mathtt{F}$, the algorithm unrolls $\mathtt{F}$ into $\mathtt{F}_1 \cup \mathtt{F}_2 \cup \mathtt{F}$. The unrolling is instrumented so as to make unfold, within the body of $\mathtt{F}_2$, only the n iterations at offsets from δ to $\delta + n - 1$. The iterations at lesser or greater offsets are left folded in $\mathtt{F}_1$ and $\mathtt{F}$ respectively. Iteration at δ, now denoting offset 0 within the body of $\mathtt{F}_2$, is potentially a promoter. In particular, any of its assignment $\mathtt{W} := \mathtt{E}$ not defining $\mathtt{W}$ in terms of itself may be the initializer of the stegomark. Given such an assignment command $\mathtt{C}$, the algorithm removes its copies at nonzero offsets within $\mathtt{F}_2$. Next, at offset $n - 1$, it looks for a unique assignment $\mathtt{C}'$ redefining $\mathtt{W}$ it terms of itself, and it applies backward-slicing using $\mathtt{C}'$ as criterion. The result is a program $\mathtt{S}$ which on input $\mathcal{I}$ first provides an initialization to $\mathtt{W}$, then updates it n times, thus computing candidate signature w. In particular, if $\mathtt{W}$ is the watermark variable, then $w = \mathfrak{s}$. Once identified $\mathfrak{s}$ among the candidates in $\mathcal{S}$, one has only to prove it to be his/her signature, as discussed above.

We now exploit the algorithm to extract signature $\mathfrak{s} = 120736$ from watermarked program $\mathtt{P}'$ of Fig. 3. Recall that in our running example the key κ is $\langle\langle \mathcal{I}_0, \mathcal{I}_1 \rangle, \delta, n\rangle = \langle\langle 8, 13\rangle, 3, 3\rangle$ and $\iota = 8$. After the unrolling of $\mathtt{F} \subseteq \mathtt{P}'$, we get program $\mathtt{Q}$ shown in Fig. 3. The promoter always covers entry points $20\mathtt{s}$, with $\mathtt{s} \in \mathbb{S}$. Here both variable $\mathtt{Y}$ and variable $\mathtt{W}$ might initialize the stegomark. However, only $\mathtt{W}$ at entry point $22\underline{\mathtt{i}}$ is able to update itself. After the slicing, we get a program $\mathtt{S} \subseteq \mathtt{Q}$ which, on input $\langle 8, 13\rangle$, sets $\mathtt{X}$ to 3, $\mathtt{Y}$ to $\mathsf{Fib}(3 + 13) = 987$ and $\mathtt{W}$ to $(215 - 987) \times 259 = -199948 = a$; just before terminating, $\mathtt{S}$ updates $n = 3$ times $\mathtt{W}$, finally getting $w = 120736 = \mathfrak{s}$.

6 Discussion

In this paper we exploit the semantics of `for`-loops to hide watermarks. Loop iterations are described extensionally by traces of execution in which iterations come one after another. When abstracted to code, they collapse into a unique loop body. Thus embedding the stegomark in the loop body means embedding it in every iteration. Our idea is to set up the stegomark so that only one iteration, the promoter, can provide the correct initialization for the computation of the signature. The choice and the localization of the promoter take place automatically thanks to loop-unrolling, used as a transformation which abstract iterations from trace to code without making them collapse. We think that our watermarking technique may be extended to other programming constructs which, like `for`-loops, provide code reuse, such as recursive functions and objects.

Signature $\mathfrak{s}$ must reliably identify the author of the watermarked program. To this end, the author can let $\mathfrak{s}$ be the product of a set of prime numbers. If some factors of $\mathfrak{s}$ are large enough, its factorization is computationally unfeasible, yet the author is able to produce it. False positives may be obtained at extraction time, both in the case of marked and unmarked loops. However, it is unlikely that their factorization is computationally unfeasible and yet known by a malicious claimer. If the extraction of the signature $\mathfrak{s}$ results in a overflow runtime error then, as suggested in [13], $\mathfrak{s}$ can be replaced with an equivalent set of smaller signatures obtained through the Chinese remainder theorem.

Watermarked programs can include more than one signature. However, they do not record which signature was inserted first, and which ones were inserted later through *additive attacks*. Unfortunately, our watermarking technique does not provide any means to register temporal precedence of signatures. To the best of our knowledge, vulnerability to additive attacks is a common drawback to all the exiting watermarking techniques [5,4]. This key problem might be solved if the insertion of the signature coincided with a not reversible semantics-preserving program evolution [3]: in such a case the order of insertion of signatures would become relevant, especially if later evolutions were strictly dependent on earlier ones. As in the field of code obfuscation [7], nontrivial semantics-preserving program transformations are likely to be systematically derived only from semantics-based frameworks. Consequently, we suppose that a better exploitation of the gap between semantics and syntax may be of help in the design of watermarking techniques that can withstand additive attacks.

Typical loop transformations [2], such as loop-reversal, loop-unrolling and loop-blocking, might distort the syntactic structure of the marked loop and obstruct the extraction of the signature; however, they are applicable only when the number of iterations can be ultimately quantified; thus a countermeasure is to embed the watermark in a `for`-loop not enjoying this property, e.g. a `for`-loop that updates an array of arbitrary length. To avoid that the inserted assigments are declared useless for the output, we *must* introduce fake dependencies between the output and `W`, for example by using opaque predicates which require hard program analyses to be removed [8]. Indeed our technique does not provide innovative contribution to the age-old problem of the resilience of watermarks. Anyway we think that semantics-based approaches may help us understand to which extent watermarks can be tied to the very core of programs.

As suggested by Fig. 1, our watermarking technique seems to resemble the DNA transcription step in protein biosynthesis. During transcription, information coded in a DNA stretch is extracted and recoded in a complementary RNA molecule. In particular, DNA unwinds and produces a small open stretch containing a promoter, which is a regulatory region providing an entry point for transcription. The transcribed RNA molecule can be partitioned in exons/introns, i.e., subregions carrying useful/useless information. Through splicing, every intron in RNA is discarded to keep only exons. Now, notice that the marked loop can be seen as the folded DNA: at extraction time, partially unrolling the marked loop corresponds to partially unwinding DNA and producing a stretch; the iteration targeted by δ is the promoter; slicing and the other minor removals correspond to RNA splicing. The idea of inserting proprietary information in a DNA molecule has been initially explored in [18]. Surely, our technique is not applicable to DNA. However, this comparison could provide intriguing insights for further research.

References

1. Apt, K.R., Plotkin, G.D.: Countable nondeterminism and random assignment. J. ACM 33(4), 724–767 (1986)
2. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Comput. Surv. 26(4), 345–420 (1994)
3. Cohen, F.B.: Operating system protection through program evolution. Comput. Secur. 12(6), 565–584 (1993)
4. Collberg, C., Carter, E., Debray, S., Huntwork, A., Kececioglu, J., Linn, C., Stepp, M.: Dynamic path-based software watermarking. SIGPLAN Not 39(6), 107–118 (2004)
5. Collberg, C., Thomborson, C.: Software watermarking: Models and dynamic embeddings. In: Principles of Programming Languages 1999, POPL 1999, San Antonio, TX (January 1999)
6. Collberg, C., Thomborson, C.: Watermarking, tamper-proofing, and obfuscation – tools for software protection. Technical Report TR00-03, University of Arizona (February 10, 2000)
7. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland (July 1997)
8. Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Principles of Programming Languages 1998, San Diego, CA (1998)
9. Cousot, P.: Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. Theoretical Computer Science 277(1-2), 47–103 (2002)
10. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angeles, California, pp. 238–252. ACM Press, New York (1977)
11. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas, pp. 269–282. ACM Press, New York (1979)
12. Cousot, P., Cousot, R.: Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In: Conference Record of the 19th ACM Symposium on Principles of Programming Languages, pp. 178–190. ACM Press, New York (2002)
13. Cousot, P., Cousot, R.: An abstract interpretation-based framework for software watermarking. In: Conference Record of the 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Venice, Italy, ACM Press, New York (2004)
14. Davidson, R.L., Myhrvold, N.: Method and systems for generating and auditing a signature for a computer program. US patent 5.559.884, Assignee: Microsoft Corporation (1996)
15. Moskowitz, S.A., Cooperman, M.: Method for stega-cipher protection of computer code. US patent 5.745.569, Assignee: The Dice Company (1996)
16. Nagra, J., Thomborson, C.D.: Threading software watermarks. In: Fridrich, J. (ed.) IH 2004. LNCS, vol. 3200, pp. 208–223. Springer, Heidelberg (2004)
17. Qu, G., Potkonjak, M.: Hiding signatures in graph coloring solutions. In: Pfitzmann, A. (ed.) IH 1999. LNCS, vol. 1768, pp. 348–367. Springer, Heidelberg (2000)
18. Shimanovsky, B., Feng, J., Potkonjak, M.: Hiding data in Dna. In: Revised Papers from the 5th International Workshop on Information Hiding, London, UK. Springer, Heidelberg (2003)
19. Weiser, M.: Program slicing. In: ICSE 1981: Proceedings of the 5th international conference on Software engineering, Piscataway, NJ, USA, pp. 439–449. IEEE Press, Los Alamitos (1981)

Inferring Min and Max Invariants Using Max-Plus Polyhedra

Xavier Allamigeon[1,3], Stéphane Gaubert[2], and Éric Goubault[3]

[1] EADS Innovation Works, SE/CS – Suresnes, France
[2] INRIA Saclay and CMAP, École Polytechnique, France
[3] CEA, LIST MeASI – Gif-sur-Yvette, France
firstname.lastname@{eads.net,inria.fr,cea.fr}

Abstract. We introduce a new numerical abstract domain able to infer min and max invariants over the program variables, based on *max-plus polyhedra*. Our abstraction is more precise than octagons, and allows to express non-convex properties without any disjunctive representations. We have defined sound abstract operators, evaluated their complexity, and implemented them in a static analyzer. It is able to automatically compute precise properties on numerical and memory manipulating programs such as algorithms on strings and arrays.

1 Introduction

We present a new abstract domain that generalizes zones [1] and octagons [2] (*i.e.* invariants of the form $\mathsf{x}_i - \mathsf{x}_j \geq c_{ij}$ and $\pm\mathsf{x}_i \pm \mathsf{x}_j \geq c'_{ij}$ respectively), while expressing a certain amount of disjunctive properties. Abstract values are max-plus polyhedra, and allow to infer relations of the form $\max(\lambda_0, \mathsf{x}_1 + \lambda_1, \dots, \mathsf{x}_n + \lambda_n) \leq \max(\mu_0, \mathsf{x}_1 + \mu_1, \dots, \mathsf{x}_n + \mu_n)$ and $\min(\lambda'_0, \mathsf{x}_1 + \lambda'_1, \dots, \mathsf{x}_n + \lambda'_n) \leq \min(\mu'_0, \mathsf{x}_1 + \mu'_1, \dots, \mathsf{x}_n + \mu'_n)$ over program variables $\mathsf{x}_1, \dots, \mathsf{x}_n$, with constants λ_i, μ_i in $\mathbb{R} \cup \{-\infty\}$ and λ'_i, μ'_i in $\mathbb{R} \cup \{+\infty\}$. For instance, the constraint $\max(\mathsf{x}, \mathsf{y}) = \max(-\infty, \mathsf{z}) = \mathsf{z}$, which forms a particular max-plus polyhedron, encodes both $\mathsf{x} - \mathsf{z} \leq 0$ and $\mathsf{y} - \mathsf{z} \leq 0$ (zone information), and "either x or y is z" (disjunctive information). Intuitively, max-plus polyhedra are the analogues of "classical" closed convex polyhedra in the max-plus algebra, which is the set $\mathbb{R} \cup \{-\infty\}$ endowed with max as additive and + as multiplicative laws.

Max-plus polyhedra encode disjunctive information, the worst-case complexity of some abstract operators may be important, although relatively similar to the case of classical polyhedra (see Theorem 1), but this disjunctive information is treated entirely semantically, and is thus fairly efficient for notoriously difficult problems in static analysis, such as proving that sorting algorithms indeed do sort (see Sect. 4). It is in the best of our knowledge the first domain whose elements describe connected but non convex sets, without having to resort to complex heuristics for building (partial) disjunctive completions.

We will use a motivating example throughout this article, which is a possible implementation of the function `memcpy` which copies exactly the first `n` characters of the string buffer `src` to `dst`:

© Springer-Verlag Berlin Heidelberg 2008

```
int i := 0;
unsigned int n, p, q;
string dst[p], src[q];
assert p >= n && q >= n;
while i <= n-1 do
  dst[i] := src[i];
  i := i+1;
done;
```

In string analysis, precise invariants over the length of the strings are needed to ensure the absence of string buffer overflows (see [3,4]). Without any information on `n`, no non-disjunctive string analysis is able to determine any precise invariant about the resulting length of the string `len_dst` (for instance, using classical polyhedra, we only get $\texttt{len\_dst} \geq 0$). Indeed, two cases have to be distinguished: (i) either `n` is strictly smaller than the source length `len_src`, so that only non-null characters are copied into `dst`, hence $\texttt{len\_dst} \geq \texttt{n}$, (ii) or $\texttt{n} \geq \texttt{len\_src}$ and the null terminal character of `src` will be copied into `dst`, thus $\texttt{len\_dst} = \texttt{len\_src}$. With our non-disjunctive analysis, we are able to infer automatically the invariant $\min(\texttt{len\_src}, \texttt{n}) = \min(\texttt{len\_dst}, \texttt{n})$, which exactly encodes the disjunction of the two cases. Besides, even with a disjunctive analysis (for example using trace partitioning [5]), it would be very complex to automatically determine the disjunction of the cases (i) and (ii), because it intrinsically relies on semantic information on strings.

Contents. Max-plus polyhedra are not new (see "Related Work" below) but have not been used yet in static analysis by abstract interpretation, and have been introduced for entirely different reasons: Section 2 is an introduction to the required results in the field. As for classical polyhedra, max-plus polyhedra can be presented both as a system of inequalities (constraints), or by a set of vertices and rays (generators), see Sections 2.2 and 2.3. But the underlying algorithms are quite different because of the structure of the max-plus algebra. For instance, unlike in classical algebra, systems of equality constraints and of inequality constraints are equivalent in $\mathbb{R}_{\max}$. In particular, the resulting abstract domains (max-plus analogues of Karr's [6] and Cousot-Halbwachs' [7]) have the same complexity. Theorem 1 is new: it gives an upper bound on the complexity of the resolution of a linear equation in the max-plus algebra, which is the cornerstone of the algorithm allowing to convert representations by (in)equalities to system of generators, and vice versa. We then prove in Sect. 2.4 that max-plus polyhedra subsume intervals and zones.

The abstraction and its semantics for a simple imperative language in terms of max-plus polyhedra are given in Section 3. It is able to infer both max and min invariants, and contains the abstract domain of octagons. We discuss linearization methods (in the sense of [2]) in Section 3.3 for abstracting in a precise manner non-linear max-plus expressions and assignments. We end up by describing some practical applications of this static analysis on strings and arrays, in Section 4. We also give a set of benchmarks based on the current implementation we made of the method.

Related Work. The max-plus analogues of convex sets were introduced by K. Zimmermann [8], who established an analogue of the separation theorem. Max-plus convex cones have been studied in idempotent analysis, after the work of

Maslov [9]. They also arise in the analysis of discrete event systems [10,11]. They have appeared recently in relation with tropical geometry and phylogenetic analysis [12]. See [13,12,14,15,16,17,18] for more background and recent developments.

Related work in abstract interpretation [19] include work on zones [1], octagons [2], (classical) polyhedra [7], disjunctive analysis [5,20,21]. An application of semirings (such as max-plus) has been made to static analysis by abstract interpretation in [22], although with different techniques and for different applications (timing behavior).

2 Max-Plus Polyhedra

2.1 The Max-Plus Semiring

The *max-plus semiring* $\mathbb{R}_{\max}$ is defined as the set $\mathbb{R} \cup \{-\infty\}$, equipped with the addition $x \oplus y := \max(x, y)$ and the multiplication $x \otimes y := x + y$. The additive law $\oplus$ is associative, commutative, and has a zero element $\mathbb{0} := -\infty$. The multiplicative law $\otimes$ is associative with a unit element $\mathbb{1} := 0$. Besides, the zero element $\mathbb{0}$ is absorbing, *i.e.* for any $x \in \mathbb{R}_{\max}$, $\mathbb{0} \otimes x = x \otimes \mathbb{0} = \mathbb{0}$. The semiring $\mathbb{R}_{\max}$ differs from a ring in that the elements are not necessarily invertible w.r.t the addition. An order $\preceq$ can be defined on $\mathbb{R}_{\max}$ by $x \preceq y \Leftrightarrow x \oplus y = y$. It coincides with the usual order on $\mathbb{R} \cup \{-\infty\}$.

The n-fold Cartesian product $\mathbb{R}^n_{\max}$ may be thought of as a space of vectors, or as an affine space of points. It can be endowed with the component-wise addition: if $\mathbf{u}, \mathbf{v}$ are two vectors in $\mathbb{R}^n_{\max}$, $\mathbf{u} \oplus \mathbf{v}$ denotes the vector whose ith component is the sum $\mathbf{u}_i \oplus \mathbf{v}_i$ of the ith components of $\mathbf{u}$ and $\mathbf{v}$. Similarly, the multiplication $\lambda\mathbf{u}$ of the vector $\mathbf{u}$ by a scalar $\lambda \in \mathbb{R}_{\max}$ is the vector of components $\lambda \otimes \mathbf{u}_i$.

Matrix operations are defined as well, by using max-plus addition and multiplication in the classical operations on matrices. Finally, given two subsets S_1 and S_2 of $\mathbb{R}^n_{\max}$, the max-plus *Minkowski sum* $S_1 \oplus S_2$ is defined as the set $\{\mathbf{x} \oplus \mathbf{y} \mid (\mathbf{x}, \mathbf{y}) \in S_1 \times S_2\}$.

2.2 Definition of Max-Plus Polyhedra Using Systems of Generators

Max-plus Convex Sets and Cones. A vector $\mathbf{x} \in \mathbb{R}^n_{\max}$ is a *max-plus linear combination* of the vectors $\mathbf{v}^1, \dots, \mathbf{v}^p \in \mathbb{R}^n_{\max}$ if $\mathbf{x} = \alpha_1\mathbf{v}^1 \oplus \dots \oplus \alpha_p\mathbf{v}^p$ for some scalars $\alpha_1, \dots, \alpha_p \in \mathbb{R}_{\max}$. The point $\mathbf{x}$ is a *max-plus convex combination* of the points $\mathbf{v}^1, \dots, \mathbf{v}^p$ if it can be written in the previous form, with the additional requirement that $\alpha_1 \oplus \dots \oplus \alpha_p = \mathbb{1}$. In the sequel, all max-plus convex or linear combinations will concern *finite* families.

A subset of $\mathbb{R}^n_{\max}$ is a *max-plus cone* if it contains all the max-plus linear combinations of its elements. Such cones may be thought of as the analogues of vector spaces or modules when the field or ring of scalars is replaced by the max-plus semiring. Hence, they have been studied under the names of *idempotent spaces* in [9] or *semimodules* in [13]. In the max-plus setting, positivity constraints

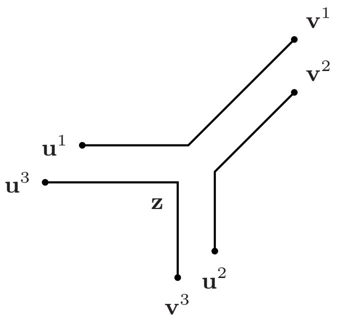


Fig. 1. The three kinds of generic max-plus segments in $\mathbb{R}^2_{\max}$

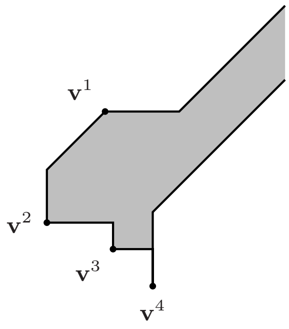


Fig. 2. A max-plus polyhedron in $\mathbb{R}^2_{\max}$ (the black border of the shape is included)

are implicit, since any scalar $\alpha \in \mathbb{R}_{\max}$ satisfies $\alpha \succeq \mathbb{0}$. For this reason, max-plus cones share many of the properties of classical convex cones.

If W is a subset of $\mathbb{R}^n_{\max}$, we denote by $\mathsf{cone}(W)$ the max-plus cone generated by W, which consists of the max-plus linear combinations of the elements of W. A max-plus cone is *finitely generated* if it can be written as $\mathsf{cone}(W)$ for some finite subset W of $\mathbb{R}^n_{\max}$. In particular, a max-plus cone generated by a (non-zero) vector is a *ray*. Such a vector is a *representative* of the ray that it generates.

Similarly, a subset of $\mathbb{R}^n_{\max}$ is a *max-plus convex* set if it contains all the max-plus convex combinations of its elements. In general, max-plus convex sets are not convex in the classical sense. We denote by $\mathsf{co}(V)$ the max-plus convex hull of V, which consists of the max-plus convex combinations of the elements of V. A set of the form $\mathsf{co}(V)$ for some finite set V is a *max-plus polytope.*

Max-plus Polyhedra. Polyhedra can be classically defined in two ways, either in terms of constraints (intersections of finitely many affine half-spaces), or in terms of vertices and rays (Minkowski sums of a polytope and of a finitely generated cone). The Minkowski-Weyl theorem proves both definitions equivalent. For the moment, let us adopt the second approach, and define a *max-plus polyhedron* to be a set of the form $P = \mathsf{co}(V) \oplus \mathsf{cone}(W)$, where V, W are finite subsets of $\mathbb{R}^n_{\max}$. The sets V and W constitute a *system of generators* of P.

Figure 2 depicts an unbounded max-plus polyhedron in $\mathbb{R}^2_{\max}$. The reader may check on the figure that this is a max-plus convex set, because it contains any segment joining two of its points (see Fig. 1 for the three kinds of max-plus segments in $\mathbb{R}^2_{\max}$).

Representing Max-plus Polyhedra by Max-plus Cones. We can represent max-plus polyhedra of $\mathbb{R}^n_{\max}$ as projections of finitely generated max-plus convex cones of $\mathbb{R}^{n+1}_{\max}$, by taking "homogeneous coordinates", as in the classical case.

Formally, if $P = \mathsf{co}(V) \oplus \mathsf{cone}(W)$, where V, W are finite subsets of $\mathbb{R}^n_{\max}$, let Z denote the subset of $\mathbb{R}^{n+1}_{\max}$ consisting of the vectors of the form $(\mathbf{v}, \mathbb{1})$ with $\mathbf{v} \in V$

or $(\mathbf{w}, \mathbb{0})$ with $\mathbf{w} \in W$, and consider the max-plus convex cone $C := \mathsf{cone}(Z)$. It is easily seen that $P = \{\mathbf{x} \in \mathbb{R}^n_{\max} \mid (\mathbf{x}, \mathbb{1}) \in C\}$. Conversely, let Z' denote any finite subset of $\mathbb{R}^{n+1}_{\max}$ such that $C = \mathsf{cone}(Z')$. After multiplying (in the max-plus sense) every element of Z' by a non-$\mathbb{0}$ scalar, we may assume that the last coordinate of every element of Z' is either $\mathbb{0}$ or $\mathbb{1}$. Then, it can be checked that $P = \mathsf{co}(V') \oplus \mathsf{cone}(W')$ where $V' = \{\mathbf{v} \mid (\mathbf{v}, \mathbb{1}) \in Z'\}$ and $W' = \{\mathbf{v} \mid (\mathbf{v}, \mathbb{0}) \in Z'\}$. This is a special case of the general correspondence between max-plus convex sets and max-plus cones which is discussed in [18].

Hence, representing max-plus polyhedra reduces to representing finitely generated max-plus cones. In the sequel, we will suppose that the representation of a max-plus polyhedron of $\mathbb{R}^n_{\max}$ by generators is a finite system G of generators of the associated max-plus cone of $\mathbb{R}^{n+1}_{\max}$. This avoids the distinction between vertices and ray representatives which may sometimes complicate the formalism, without loss of generality.

Membership to a Finitely Generated Max-plus Cone. Let G denote a set of p vectors of $\mathbb{R}^{n+1}_{\max}$. Testing whether a vector $\mathbf{x}$ is in $\mathsf{cone}(G)$ is equivalent to determine whether the system of equations $G\mathbf{y} = \mathbf{x}$ admits a solution. We use here the same notation, G for the matrix the columns of which are the elements of the generating set. The equation $G\mathbf{y} = \mathbf{x}$ may not have a solution, but the inequality $G\mathbf{y} \preceq \mathbf{x}$ always does. Besides, if it is interpreted in the completion of the max-plus semiring, it admits a maximal solution, denoted by $G \setminus \mathbf{x}$ and given by the following residuation formula: $(G \setminus \mathbf{x})_j := \min_{1 \le i \le n+1}(\mathbf{x}_i - G_{ij})$, with the convention $-\infty + \infty = +\infty$. If G has no column identically $-\infty$, $G \setminus \mathbf{x}$ belongs to $\mathbb{R}^p_{\max}$ for all $\mathbf{x} \in \mathbb{R}^{n+1}_{\max}$. It follows that $G\mathbf{y} = \mathbf{x}$ has a solution $\mathbf{y} \in \mathbb{R}^p_{\max}$ if and only if $G(G \setminus \mathbf{x}) = \mathbf{x}$, a test which can be done in $O(np)$ operations. More details can be found in [17].

Minimal Systems of Generators. The representation of P by a system of generators is not unique, but as for classical polyhedra, there is a minimal representation, involving sets of vectors having certain extremality properties in the associated cone C. A vector $\mathbf{w}$ is an *extreme generator* of a max-plus cone C if $\mathbf{w} \in C$, and $\mathbf{w}$ cannot be written as the max-plus sum of two vectors of C that are both different from it. Then, the set of scalars multiples of $\mathbf{w}$ is an *extreme ray* of C. It is known that a finitely generated max-plus cone is generated by its extreme rays. It follows that C has a non redundant generating family, which is unique up to a normalization of its elements, obtained by selecting one representative in each extreme ray of C. This family forms a minimal representation of P by generators. The reader can refer to [18,17] for recent accounts and refinements of this result.

To compute the extreme rays of C, we start from any generating system G, assuming that it contains no proportional vectors, and we eliminate any vector of the family which is a max-plus linear combination of the other ones (see the previous paragraph). If G consists of p generators, this can be done in $O(n \times p^2)$ operations. Some additional algorithmic information can be found in [17].

As in the classical case, we can find max-plus polyhedra of $\mathbb{R}^2_{\max}$ with an arbitrarily large number of extreme points.

2.3 Definition of Max-Plus Polyhedra by Systems of Constraints

An important similarity of max-plus polyhedra with classical ones is that they can be equivalently defined as the solutions of systems of constraints. Each constraint consists of an inequality of the form $\mathbf{ax} \oplus b \succeq \mathbf{cx} \oplus d$, where $\mathbf{x} \in \mathbb{R}^n_{\max}$, $\mathbf{a}, \mathbf{c} \in \mathbb{R}^{1\times n}_{\max}$ and $b, d \in \mathbb{R}_{\max}$.

However, in $\mathbb{R}^n_{\max}$, systems of equality and inequality constraints are equivalent. Indeed, any inequality can be written as an equality since $y \succeq z \Leftrightarrow y = y \oplus z$. As a consequence, systems of inequality and equality constraints have the same expressiveness, and in particular, inferring invariants involving equality constraints is as difficult as inferring inequality invariants. We chose to use here systems of equality constraints.

Solutions of a System of Constraints. The solutions of a homogeneous system of equations $A\mathbf{x} = B\mathbf{x}$, were first studied in [23]. In particular, the following proposition was proven (see also [10]):

Proposition 1. *The solutions of the homogeneous system $A\mathbf{x} = B\mathbf{x}$ of $\mathbb{R}^n_{\max}$, where $A, B \in \mathbb{R}^{s\times n}_{\max}$, form a finitely generated max-plus cone.*

More generally, a non-homogeneous system $A\mathbf{x} \oplus \mathbf{b} = C\mathbf{x} \oplus \mathbf{d}$ can be associated to the homogeneous system $\begin{pmatrix} A & \mathbf{b} \end{pmatrix} \mathbf{z} = \begin{pmatrix} C & \mathbf{d} \end{pmatrix} \mathbf{z}$ of $\mathbb{R}^{n+1}_{\max}$. Then, the solutions of the former are given by the first n coordinates of the solutions $\mathbf{z}$ verifying $\mathbf{z}_{n+1} = \mathbb{1}$ of the latter. Using the equivalence between max-plus polyhedra and cones established in Sect. 2.2, the following statement holds:

Corollary 1. *The solutions of the system of equations $A\mathbf{x} \oplus \mathbf{b} = C\mathbf{x} \oplus \mathbf{d}$, where $A, B \in \mathbb{R}^{s\times n}_{\max}$ and $\mathbf{b}, \mathbf{d} \in \mathbb{R}^s_{\max}$, form a max-plus polyhedron of $\mathbb{R}^n_{\max}$.*

In particular, a representation of the solution polyhedron can be obtained by computing a minimal system of generators of the cone of solutions of $\begin{pmatrix} A & \mathbf{b} \end{pmatrix} \mathbf{z} = \begin{pmatrix} C & \mathbf{d} \end{pmatrix} \mathbf{z}$. This is why we only consider homogeneous systems $E\mathbf{z} = F\mathbf{z}$ in $\mathbb{R}^{n+1}_{\max}$ for the rest of the section.

First, let us consider the case in which the system of constraints is reduced to one equation $\mathbf{ez} = \mathbf{fz}$, where $\mathbf{e}, \mathbf{f} \in \mathbb{R}^{1\times(n+1)}_{\max}$. We denote by $(\epsilon^i)_{1\leq i\leq n+1}$ the max-plus analogue of a canonical basis in $\mathbb{R}^{n+1}_{\max}$, *i.e.* $\epsilon^i_i = \mathbb{1}$ and $\epsilon^i_j = \mathbb{0}$ for $j \neq i$. It can be shown that the vectors $(\mathbf{f}_j \epsilon^i) \oplus (\mathbf{e}_i \epsilon^j)$, where $\mathbf{e}_i \succeq \mathbf{f}_i$ and $\mathbf{e}_j \preceq \mathbf{f}_j$, form a generating system of the solution cone.

The general case of a system of s equations can be solved by induction on s, following the method by *elimination* proposed in [23]:

- when $s = 0$, there is no constraint, so that the family ϵ^i form a generating system of the solution.
- if $s \geq 1$, let $E'\mathbf{z} = F'\mathbf{z}$ be the system of equations formed by the $(s-1)$ first equations, and $\mathbf{ez} = \mathbf{fz}$ the last equation of $E\mathbf{z} = F\mathbf{z}$. If $G' = (\mathbf{g}^1, \ldots, \mathbf{g}^p)$ is a system of generators of $E'\mathbf{z} = F'\mathbf{z}$, then we have $E\mathbf{z} = F\mathbf{z}$ if and only if there exists $\mathbf{y} \in \mathbb{R}^p_{\max}$ such that $(\mathbf{e}G')\mathbf{y} = (\mathbf{f}G')\mathbf{y}$ and $\mathbf{z} = G'\mathbf{y}$ (G' being seen as a matrix whose columns are the $\mathbf{g}^i$). The equation $(\mathbf{e}G')\mathbf{y} = (\mathbf{f}G')\mathbf{y}$

of $\mathbb{R}^p_{\max}$ can be solved using the method given above. If $H = (\mathbf{h}^1, \ldots, \mathbf{h}^q)$ is a generating system of its solutions, the vectors $G'\mathbf{h}^1, \ldots, G'\mathbf{h}^q$ form a system of generators of the solutions of $E\mathbf{z} = F\mathbf{z}$.

Theorem 1. *A minimal system of generators of the solutions of the s equations $E\mathbf{z} = F\mathbf{z}$ in $\mathbb{R}^{n+1}_{\max}$ can be computed in $O(n \times s \times c^4_{n+1,s})$ operations, where $c_{n+1,s}$ is the maximal number of generators of the set of solutions of a system of s equations in $\mathbb{R}^{n+1}_{\max}$.*

As a consequence, the solving algorithm is polynomial in the maximal number of the generators which may arise. In comparison, the cost of Chernikova's algorithm [24], which allows to convert the representation of a classical convex polyhedron by inequalities into an equivalent representation by generators, is quadratic in the number of generators, when it is executed on hypercubes of $\mathbb{R}^n$. However, it seems that the problem of solving max-plus (in)equalities is intrinsically more complex, as even an inequality $\mathbf{ex} \succeq \mathbf{fx}$ of $\mathbb{R}^n_{\max}$ generates a quadratic number of generators in n, while this number is linear in the classical case. Moreover, most implementations of the domain of classical convex polyhedra now involve efficient tests based on the number of inequalities saturated by the computed generators, in order to eliminate redundant ones (see, for instance, [25]). In contrast, no such properties relative to the saturation of max-plus inequalities are yet proven.

Bounding the maximal number of generators $c_{n+1,s}$ is an interesting combinatorial problem. An exponential bound is given in [26], where it is shown that $c_{n+1,s} \leq s \times \left(n^2/3 + n + 1\right)^s$, but the optimal bound is not known. Future work could focus on the comparison of $c_{n+1,s}$ with its classical analogue for convex polyhedra, which is in $O(s^n)$. But for now, all we can say is that the solution of a system can be computed in $O(s^2 \times n^{8s+1})$ operations.

Example 1. With $n = 2$, let us consider the system of two equations $\mathbf{x}_1 \oplus 1 = \mathbf{x}_1$ and $\mathbf{x}_2 \oplus 1 = \mathbf{x}_2$. It corresponds to the system $\mathbf{x}_1 \geq 1$ and $\mathbf{x}_2 \geq 1$. The associated homogeneous system is $\mathbf{z}_1 \oplus 1\mathbf{z}_3 = \mathbf{z}_1$ and $\mathbf{z}_2 \oplus 1\mathbf{z}_3 = \mathbf{z}_2$. Solving the first equation yields a generating family G consisting of the vectors $\mathbf{g}^1 = [1; -\infty; 0]$, $\mathbf{g}^2 = [-\infty; 0; -\infty]$, $\mathbf{g}^3 = [0; -\infty; -\infty]$. Multiplying the left and the right members of the second equation by the matrix G yields the equation $1\mathbf{y}_1 \oplus \mathbf{y}_2 = \mathbf{y}_2$, whose generating family H consists of the vectors $\mathbf{h}^1 = [-\infty; 0; -\infty]$, $\mathbf{h}^2 = [-\infty; -\infty; 0]$, and $\mathbf{h}^3 = [0; 1; -\infty]$. The vectors $G\mathbf{h}^1$, $G\mathbf{h}^2$, and $G\mathbf{h}^3$ form the family $([-\infty; 0; -\infty], [0; -\infty; -\infty], [1; 1; 0])$, which is obviously minimal. It represents a max-plus polyhedron with one vertex $[1; 1]$, and two rays with representatives $[0; -\infty]$ and $[-\infty; 0]$.

From Systems of Generators to Systems of Constraints. A system of generators can be converted to a system of constraints describing the same max-plus polyhedron. Given a max-plus polyhedron P provided with a system of generators G, the set $P^\perp$ of constraints $\mathbf{ax} \oplus b = \mathbf{cx} \oplus d$ verified by the polyhedron P is a cone of $(\mathbb{R}^{1\times n}_{\max} \times \mathbb{R}_{\max})^2$ (each constraint $\mathbf{ax} \oplus b = \mathbf{cx} \oplus d$ being represented by the pair $((\mathbf{a}, b), (\mathbf{c}, d))$). Moreover, it can be shown that a constraint is verified by the polyhedron if and only if it is verified by all its generators. Hence, we have $P^\perp = \{((\mathbf{a}, b), (\mathbf{c}, d)) \in (\mathbb{R}_{\max}^{1\times n} \times \mathbb{R}_{\max})^2 \mid \forall i.\, \begin{pmatrix}\mathbf{a} & b\end{pmatrix} \mathbf{g}^i = \begin{pmatrix}\mathbf{c} & d\end{pmatrix} \mathbf{g}^i\}$, *i.e.*

$$P^\perp = \left\{((\mathbf{a}, b), (\mathbf{c}, d)) \in (\mathbb{R}_{\max}^{1\times n} \times \mathbb{R}_{\max})^2 \mid {}^tG \begin{pmatrix} {}^t\mathbf{a} \\ b \end{pmatrix} = {}^tG \begin{pmatrix} {}^t\mathbf{c} \\ d \end{pmatrix} \right\} ,$$

where ${}^t\cdot$ is the matrix transposition operator.

As a consequence, a minimal system of generators of the cone $P^\perp$ can be computed by using the algorithm presented in the previous paragraph, with a complexity in $O(p \times n \times c_{2n+2,p}^4)$. Then, the system of constraints formed by the generators of the cone $P^\perp$ can be shown to be a representation of the max-plus polyhedron P under the form of constraints. As for generators, we are interested in manipulating minimal representations by systems of constraints. Here, the computed constraints form a minimal generating family of the cone $P^\perp$, which is a good point. However, it may not be a minimal system of constraints, *i.e.* some constraints are possibly redundant. This is basically due to the fact that the cone $P^\perp$ represents a set of equations closed by symmetry, reflexivity, and transitivity. Extracting a minimal system of constraints would have a major drawback: it would be very costly since we would have to compare corresponding max-plus polyhedra, hence to convert many systems of constraints to generators (see the definition of the abstract partial order in Sect. 3.1). Moreover, in our experimentations (see Sect. 4), the size taken by systems of constraints is negligible, this is why minimal generating families of the cone $P^\perp$ are satisfactory.

2.4 Max-Plus Polyhedra and Zones

Interval and zone constraints are obviously particular forms of max-plus systems of constraints described in Sect. 2.3.

We next show that a representation by intervals and zones can be extracted from a system of generators of a max-plus polyhedron. Recall that if $A \in \overline{\mathbb{R}}_{\max}^{n\times p}$, the residuated matrix [27] A/A is given by $(A/A)_{ij} = \min_{1\leq k\leq p} A_{ik} - A_{jk}$ (with $-\infty + \infty = +\infty$). Observe that if G is a minimal system of generators of $\mathbb{R}_{\max}^{n+1}$, and if G (seen as a matrix) does not have a row consisting only of $-\infty$ entries, then $G/G \in \mathbb{R}_{\max}^{(n+1)\times(n+1)}$. Using the fact that $\mathsf{cone}(G/G)$ (*i.e.* the cone generated by the column of G/G) is the least sublattice containing $\mathsf{cone}(G)$ [28], we deduce the following theorem:

Theorem 2. *The cone* $\mathsf{cone}(G/G)$ *coincides with the zone of* $\mathbb{R}_{\max}^n$ *defined by:*

$$\forall i, j \in \{1, \dots, n\},\ \mathbf{x}_i - \mathbf{x}_j \geq (G/G)_{ij} ,$$
$$\forall i \in \{1, \dots, n\},\ (G/G)_{i,n+1} \leq \mathbf{x}_i \leq -(G/G)_{n+1,i} .$$

Moreover, if G has no row consisting only of $-\infty$ entries, then the smallest zone containing the $\mathsf{cone}(G)$ *is given by* $\mathsf{cone}(G/G)$*.*

During the reduction to zones, the rows of G consisting only of $-\infty$ entries can be simply not considered, since they correspond to coordinates $\mathbf{x}_i$ constant equal to $-\infty$. Hence, Th. 2 provides an effective algorithm to convert max-plus polyhedra to zones.

3 Abstract Semantics

Let us consider a set Var of n distinct variables x_i. Our abstraction consists in representing sets of environments $\sigma : \mathsf{Var} \to \mathbb{R}$ by max-plus polyhedra P of $\mathbb{R}^{2n}_{\max}$, *i.e.* either by minimal systems G of generators of $\mathbb{R}^{2n+1}_{\max}$, or by systems S of max-plus equality constraints $A\mathbf{x} \oplus \mathbf{b} = C\mathbf{x} \oplus \mathbf{d}$ of $\mathbb{R}^{2n}_{\max}$.[1] Intuitively, the n first dimensions of the polyhedra represent the variables x_i, while the n last ones represent their opposite $-\mathsf{x}_i$. The concretization of max-plus polyhedra is defined equivalently according to their representation:

$$\gamma(P) := \{\sigma \mid (\sigma(\mathsf{x}_1), \dots, \sigma(\mathsf{x}_n), -\sigma(\mathsf{x}_1), \dots, -\sigma(\mathsf{x}_n), \mathbb{1}) \in \mathsf{cone}(G))\} \quad ,$$

$$\text{or} \quad \gamma(P) := \left\{\sigma \mid \begin{array}{l} A(\sigma(\mathsf{x}_1), \dots, \sigma(\mathsf{x}_n), -\sigma(\mathsf{x}_1), \dots, -\sigma(\mathsf{x}_n)) \oplus \mathbf{b} \\ \quad = C(\sigma(\mathsf{x}_1), \dots, \sigma(\mathsf{x}_n), -\sigma(\mathsf{x}_1), \dots, -\sigma(\mathsf{x}_n)) \oplus \mathbf{d} \end{array}\right\} \quad .$$

As mentioned in Sect. 1, this allows to infer invariants of the form $\max(\lambda_0, \mathsf{x}_1 + \lambda_1, \dots, \mathsf{x}_n + \lambda_n) \leq \max(\mu_0, \mathsf{x}_1 + \mu_1, \dots, \mathsf{x}_n + \mu_n)$ and $\min(\lambda'_0, \mathsf{x}_1 + \lambda'_1, \dots, \mathsf{x}_n + \lambda'_n) \leq \min(\mu'_0, \mathsf{x}_1 + \mu'_1, \dots, \mathsf{x}_n + \mu'_n)$.[2]

3.1 Order-Theoretic Operators

Partial Order. An abstract partial order can be defined on max-plus polyhedra by comparing systems of generators. Given two max-plus polyhedra P and Q represented by the systems of generators G and H respectively, we have $P \sqsubseteq Q$ if and only if for any $\mathbf{g} \in G$, $\mathbf{g} \in \mathsf{cone}(H)$ (or equivalently, $H(H \setminus \mathbf{g}) = \mathbf{g}$). Hence, the concretization γ can be shown to be monotonic. The complexity of the evaluation of $G \sqsubseteq H$ is $O(n \times p \times q)$, p and q being the cardinality of the families G and H. Note that we can define equivalently $\sqsubseteq$ by using a representation of Q by a system of constraints $A\mathbf{x} \oplus \mathbf{b} = C\mathbf{x} \oplus \mathbf{d}$, and testing whether $\begin{pmatrix} A & \mathbf{b} \end{pmatrix} \mathbf{g} = \begin{pmatrix} D & \mathbf{d} \end{pmatrix} \mathbf{g}$ for any $\mathbf{g}$ in G. Then, the operation has a complexity in $O(n \times p \times t)$, where t is the number of constraints in the system of Q.

Joining Max-plus Polyhedra. Given two systems of generators G and H, an abstract join operator $\sqcup$ can be defined as the minimal system G of generators extracted from the family $G \cup H$. If p and q are the cardinality of the families G and H, the union can be performed in $O(n \times (p+q)^2)$. It can be shown to be a sound join operator, and even the best possible one.

Intersection. By duality, an abstract intersection operator can be naturally defined on two polyhedra by concatenating the systems of constraints representing the polyhedra. Equivalently, we can define the intersection operator when one polyhedron is represented by a minimal system G of generators while the other is

[1] Each system S is represented by a minimal generating family of the cone $P^\perp$.

[2] The latter are computed under the form $\max(-\lambda'_0, -\mathsf{x}_1 - \lambda'_1, \dots, -\mathsf{x}_n - \lambda'_n) \geq \max(-\mu'_0, -\mathsf{x}_1 - \mu'_1, \dots, -\mathsf{x}_n - \mu'_n)$. In fact, the abstract domain is able to infer more general invariants of the form $\max(\lambda_0, \mathsf{x}_1 + \lambda_1, \dots, \mathsf{x}_n + \lambda_n, -\mathsf{x}_1 - \lambda'_1, \dots, -\mathsf{x}_n - \lambda'_n) \leq \max(\mu_0, \mathsf{x}_1 + \mu_1, \dots, \mathsf{x}_n + \mu_n, -\mathsf{x}_1 - \mu'_1, \dots, -\mathsf{x}_n - \mu'_n)$.

represented by a system of constraints $A\mathbf{x} \oplus \mathbf{b} = C\mathbf{x} \oplus \mathbf{d}$. Indeed, it can be shown that solving the homogeneous system of constraints $\big(\big(A\ \mathbf{b}\big)\, G\big)\, \mathbf{z} = \big(\big(C\ \mathbf{d}\big)\, G\big)\, \mathbf{z}$ by replacing the family ϵ^i by the $\mathbf{g}^i$ in the initial step, exactly yields a minimal system of generators of the intersection. In that case, the complexity of the intersection is $O(t \times p \times c_{p,t}^4)$, where p is the cardinality of G and t the number of constraints of the system $A\mathbf{x} \oplus \mathbf{b} = C\mathbf{x} \oplus \mathbf{d}$.

The intersection operator allows to handle conditional program statements of the form $\pm\mathsf{x} \leq k$ or $\pm\mathsf{x} \leq \pm\mathsf{y} + k$. Other conditions can be either ignored (which is sound), or handled using a linearization (see Sect. 3.3).

Widening. If $n \geq 2$, infinite ascending chains of max-plus polyhedra of $\mathbb{R}_{\max}^n$ can be built. As a result, a widening operator is defined to enforce convergence. It follows the initial definition of the widening over classical convex polyhedra [7]. Formally, if two max-plus polyhedra P and Q are respectively represented by a system of constraints and a minimal system G of generators, the max-plus polyhedron $P \mathbin{\nabla} Q$ is defined as the system of constraints formed by the constraints $\mathbf{ax} \oplus b = \mathbf{cx} \oplus d$ of P which are also verified by Q, *i.e.* $\forall \mathbf{g} \in G.\ \big(\mathbf{a}\ b\big)\, \mathbf{g} = \big(\mathbf{c}\ d\big)\, \mathbf{g}$.

As for the widening defined in [7], the result of the widening depends on the system of constraints chosen to represent the max-plus polyhedra. In [29], the definition of the widening over classical convex polyhedra was improved to overcome this problem, by adding to the result the constraints of Q which are equivalent to some constraints of P. They can be discovered either by checking whether they can replace a constraint of P without changing the represented polyhedron, or by considerations on the saturation of some linear inequalities by generators [30]. In max-plus algebra, the former method is particularly costly since it requires to convert some systems of constraints to generators. Moreover, we do not have yet any proof that the latter approach could be applied, because the equivalence between inequalities and equalities in the max-plus algebra makes the problem harder. For that reason, the actual definition of the widening operator is not fully satisfactory. Nevertheless, the experimentations are very encouraging since the widening allows to exactly infer the expected invariant for each of our examples (see Sect. 4).

Reduction. A system of octagonal constraints (*i.e.* of the form $\pm\mathsf{x}_i \pm \mathsf{x}_j \geq c_{ij}$) can be extracted from any representation by generators using Sect. 2.4. These constraints can be then refined using the closure algorithm of octagons [2]. The resulting octagon can be seen as a max-plus polyhedron over the variables $\pm\mathsf{x}_i$. Intersecting it with the initial max-plus polyhedron yields a smaller abstract element w.r.t $\sqsubseteq$, but which represents the same set of concrete states. This defines a reduction operator, which allows our representation to be more precise than the abstract domain of octagons. Intuitively, the reduction operator enables a communication between the variables x_i and $-\mathsf{x}_i$. Note that, as for octagons, the convergence property of the widening operator may not hold if the reduction operator is applied to widened max-plus polyhedra.

3.2 Assignments

Max-plus Assignments. A *max-plus assignment* is an assignment of the form $\mathbf{x}_i \leftarrow \left(\oplus_{j=1}^{n} m_j \mathbf{x_j}\right) \oplus m_{n+1}$, for some $1 \leq i \leq n$, and $m_1, \ldots, m_n, m_{n+1} \in \mathbb{R}_{\max}$. In particular, max-plus assignments include operations $\mathbf{x}_i \leftarrow m_{n+1}$ and $\mathbf{x}_i \leftarrow \mathbf{x}_j + m_j$ (where $+$ is the classical addition). The abstract operator for max-plus assignments consists in multiplying a minimal system of generators by a matrix M corresponding to the assignment: M coincides with the max-plus identity matrix, except that its ith row is replaced by $m_1 \ldots m_{n+1}$. It then remains to extract a minimal system of generators from the result. This operator can be shown to be sound. Its cost is $O(n \times (n^2 + p^2))$. It can be easily generalized to handle parallel max-plus assignments, without changing the complexity. Thus, program assignments of the form $\mathsf{x} \leftarrow k$ and $\mathsf{x} \leftarrow \pm\mathsf{y} + k$ are implemented as parallel max-plus assignments on the dimensions of variables x and $-\mathsf{x}$.

Non-deterministic Assignments. A non-deterministic assignment $\mathbf{x}_i \leftarrow ?$ can be handled by adding a representative $\mathbf{h}$ of the ray formed by the ith axis (*e.g.* $\mathbf{h}_i = \mathbb{1}$ and $\mathbf{h}_j = \mathbb{0}$) to a minimal system of generators, and then extracting a new minimal system from the resulting family. This defines a sound operator, whose cost is in $O(n \times p^2)$ (p being the size of the initial system of generators).

Some assignments which do not belong to the classes previously discussed can be linearized (see Sect. 3.3). Other can be soundly treated as non-deterministic.

3.3 Linearization

In this section, we indicate how to interpret general linear assignments (as in classical linear algebra), *i.e.* non-linear max-plus expressions. For sake of simplicity, the description is restricted to max-plus polyhedra of $\mathbb{R}_{\max}^n$, which infer information on the positive variables $\mathsf{x}_1, \ldots, \mathsf{x}_n$. A generalization to the full abstraction including the opposites $-\mathsf{x}_1, \ldots, -\mathsf{x}_n$ is straightforward.

Suppose the variables $\mathsf{x}_1, \ldots, \mathsf{x}_n$, at some control point of a program, belong to a max-plus polyhedron $P = \mathsf{co}(V) \oplus \mathsf{cone}(W)$. If $V = (\mathbf{v}^i)_i$ and $W = (\mathbf{w}^j)_j$, then for any k, $\mathsf{x}_k = \left(\bigoplus_{i=1}^{p} \alpha_i \mathbf{v}_k^i\right) \oplus \left(\bigoplus_{j=1}^{q} \beta_j \mathbf{w}_k^j\right)$, with $\bigoplus_{i=1}^{p} \alpha_i = \mathbb{1}$. In particular, $\alpha_i \preceq \mathbb{1}$ for $i \in \{1, \ldots, p\}$. More than that, we have $\alpha_i = \mathbb{1}$ for some $i \in \{1, \ldots, p\}$, hence $\bigoplus_{i=1}^{p} \alpha_i \mathbf{v}_k^i \succeq \min_{i=1}^{p} \mathbf{v}_k^i$. Hence, we can write equivalently: $\mathsf{x}_k = \mathbf{v}_k^0 \oplus \bigoplus_{i=1}^{p} \alpha_i \mathbf{v}_k^i \oplus \bigoplus_{j=1}^{q} \beta_j \mathbf{w}_k^j$, with $\bigoplus_{i=1}^{p} \alpha_i = \mathbb{1}$ and $\mathbf{v}_j^0 = \min_{i=1}^{p} \mathbf{v}_j^i$.

Sum. Consider now the assignment $\mathsf{x}_{n+1} \leftarrow \mathsf{x}_k + \mathsf{x}_l$ ($+$ is here the standard addition, *i.e.* the multiplication in the max-plus algebra), where x_{n+1} is a newly introduced variable (up to assigning x_{n+1} to a variable x_m later, and removing the $(n+1)$-th dimension). We then have:

$$\mathsf{x}_{n+1} = \mathbf{v}_k^0 \mathbf{v}_l^0 \oplus \bigoplus_{i=1}^{p} \alpha_i \left(\mathbf{v}_k^i \mathbf{v}_l^0 \oplus \mathbf{v}_k^0 \mathbf{v}_l^i\right) \oplus \bigoplus_{j=1}^{q} \beta_j \left(\mathbf{w}_k^j \mathbf{v}_l^0 \oplus \mathbf{v}_k^0 \mathbf{w}_l^j\right) \oplus N \enspace ,$$

which can be recognized as the sum of the first-order Taylor expansion (or linearization) of the function $(x, y) \mapsto xy$ (in the max-plus algebra), and a non-linear residual term N.

Let us define some new generators: $\mathbf{v}'^0 = \big(\mathbf{v}^0, \mathbf{v}_k^0\mathbf{v}_l^0\big)$, $\mathbf{v}'^i = (\mathbf{v}^i, \mathbf{v}_k^i\mathbf{v}_l^0 \oplus \mathbf{v}_k^0\mathbf{v}_l^i)$ for $i = 1, \dots, p$, and $\mathbf{w}'^i = (\mathbf{w}^i, \mathbf{w}_k^i\mathbf{v}_l^0 \oplus \mathbf{v}_k^0\mathbf{w}_l^i)$ for $i = 1, \dots, q$. Besides, we add up a vertex, abstracting the first part of the non-linear term N: $\mathbf{v}^{p+1} = [\mathbb{0}; \dots; \mathbb{0}; \bigoplus_{i,j=1}^{p} \mathbf{v}_k^i\mathbf{v}_l^j]$, and if $q > 0$, a ray abstracting the second part of the non-linear term N: $\mathbf{w}^{q+1} = [\mathbb{0}; \dots; \mathbb{0}; \mathbb{1}]$. The returned system of generators represents a sound approximation of the assignment on the initial polyhedron.[3]

Multiplication by a Constant. We interpret now the assignment $\mathsf{x}_{n+1} \leftarrow a \times \mathsf{x}_k$ where a is a constant. We suppose $a \geq 0$.[4] Then we have $\mathsf{x}_{n+1} = a \times \mathbf{v}_k^0 \oplus \bigoplus_{i=1}^{p}(a \times \alpha_i)(a \times \mathbf{v}_k^i) \oplus \bigoplus_{j=1}^{q}(a \times \beta_j)(a \times \mathbf{w}_k^j)$,. Except in the trivial case $a = 1$, we cannot abstract very precisely this expression. Our only choice is to introduce a new vertex $\mathbf{v}^{p+1} = [\mathbb{0}; \dots; \mathbb{0}; \bigoplus_{i=1}^{p} a \times \mathbf{v}_k^i \oplus \bigoplus_{j=1}^{q} a \times \mathbf{w}_k^j]$.

Comparison with Linearization in Octagons. Ordinarily, only $\mathsf{x} \leftarrow \pm\mathsf{y} + [a, b]$, $\mathsf{x} \leftarrow [a, b]$, or $\mathsf{x} \leftarrow \pm\mathsf{x} + [a, b]$ are interpreted exactly on octagons [2]. We claim that given $\mathsf{z} \leftarrow [a, b]$ encoded as a max-plus polyhedron, $\mathsf{x} \leftarrow \pm\mathsf{y} + \mathsf{z}$, $\mathsf{x} \leftarrow \mathsf{z}$, and $\mathsf{x} \leftarrow \pm\mathsf{x}+\mathsf{z}$ are interpreted with our linearization, exactly as an octagon would do. We also claim that linearization of assignments in the style of [31] for octagons encoded as max-plus polyhedra is in general as precise or less precise than our linearization on these max-plus polyhedra. This will be developed elsewhere.

4 Examples and Benchmarks

The abstraction defined in Sect. 3 has been implemented in an analyzer of 3 500 lines of OCaml. Our prototype only manipulates systems of generators, except in the widening steps for which conversions to constraints are needed. It has been evaluated on various programs described below.[5] Table 1 indicates the number of lines and variables of each program, the time the analyzer needs to compute invariants, and the number of generators the resulting invariants have. For each example, the memory consumption is negligible (at worst 9 Mb for `oddeven9`).

String and Array Manipulation. Our analyzer is able to infer precise invariants on the advanced string manipulating functions `memcpy` and `strncpy`. The latter copies at most `n` characters from `src` into `dst`. In particular, if the length of `src` is smaller that `n`, the remainder of `dst` is filled with null characters. For both programs, the expected final invariant $\min(\mathtt{len\_src}, \mathtt{n}) = \min(\mathtt{len\_dst}, \mathtt{n})$ is successfully discovered by our analyzer.

The program `partd` is a decrementing initialization program which fills an array with a value `c`, from the index `q` to $\mathtt{p} + 1$. Some array analyzes [32,33] allow to infer the loop invariant $\mathbf{c}(\mathtt{i} + 1, \mathtt{q})$, which means that the array `t` contains the value `c` between the indexes $\mathtt{i} + 1$ and `q` (both included). However,

[3] Albeit not detailed here, the case $p = 1$ can be handled in a more precise manner.

[4] The case $\mathsf{x}_{n+1} \leftarrow a \times \mathsf{x}_k$ with $a \leq 0$ rewrites into $-\mathsf{x}_{n+1} = (-a) \times \mathsf{x}_k$ with $-a \geq 0$.

[5] Source codes are available at `http://www.lix.polytechnique.fr/~allamige`.

Table 1. Analysis benchmarks on a 3 GHz Pentium with 4 Gb RAM

Program	# lines	# var.	# time (s)	# gen.
memcpy	9	6	2.37	7
strncpy	19	7	9.82	8
parti	8	3	0.008	3
partd	7	3	0.008	3
partd2	10	4	0.084	4
partd3	14	5	0.272	5
partd4	17	6	0.73	6
partd5	20	7	2.15	7
partd6	22	8	6.98	8
partd7	25	9	10.98	9
partd8	28	10	28.35	10
partd9	31	11	40.95	11
partd10	34	12	68.94	12

Program	# lines	# var.	# time (s)	# gen.
partd11	37	13	129.86	13
partd12	40	14	196.32	14
partd13	43	15	335.62	15
partd14	46	16	420.03	16
partd15	49	17	672.49	17
bubble3	21	7	0.016	6
oddeven3	28	7	0.012	8
oddeven4	39	9	0.06	16
oddeven5	70	11	1.13	32
oddeven6	86	13	7.76	64
oddeven7	102	15	34.42	116
oddeven8	118	17	178.42	196
oddeven9	214	19	25939.76	512

without prior information on the order of `p` and `q`, the final invariant on `i` with classical convex polyhedra is only $\mathtt{i} \leq \mathtt{p} \wedge \mathtt{i} \leq \mathtt{q}$. Our analyzer is able to discover the relations $\mathtt{i} = \min(\mathtt{p}, \mathtt{q})$, which is the most precise invariant. Similarly, each program `partd`k corresponds to a sequence of k partial initializations. For each, our analyzer infers the expected invariant expressing that `i` is the minimum of the $k + 1$ indexes. An incrementing version of `partd`, `parti`, is also successfully analyzed.

For these examples, the widening steps (which require conversions from generators to constraints) are by far the more time consuming steps.

Sorting Algorithms. Consider now an implementation of bubble sort. We completely unfold the loop and specialize it to an array of three elements $[\mathtt{x}, \mathtt{y}, \mathtt{z}]$. The resulting sorted array is supposed to be $[\mathtt{i}, \mathtt{j}, \mathtt{k}]$. Our analyzer proves in particular that `i` and `k` are respectively the smallest and biggest elements of the three initial ones, without resorting to a heavy disjunctive analysis. In order to prove more, *i.e.* about `j` (we "only" get $\mathtt{j} \geq \mathtt{i}$ and $\mathtt{j} \leq \mathtt{k}$, and `j` is less than the maximum of any pair of entries in the input array), we would need mixed constraints with min and max (see Section 5).

Last but not least, consider the odd-even sort [34] on 2^k elements. Here we start with an array of four elements $[\mathtt{i}, \mathtt{j}, \mathtt{k}, \mathtt{l}]$, the resulting sorted array should be $[\mathtt{x}, \mathtt{y}, \mathtt{z}, \mathtt{t}]$. We find automatically in particular that `x` and `t` are the minimum and the maximum of `i`, `j`, `k`, and `l`. In fact, the max-plus constraints that are generated are quite dense. This sorting algorithm is probably of worst-case complexity for our analysis. Programs `oddeven`i (for $i = 3$ to 9) are odd-even sorting algorithms for i elements. The analyzer proves that the last (resp. first) element of the resulting array is the maximum (resp. minimum) of the inputs. The exponentially growing complexity of the analysis is mainly due to the intersection operations (corresponding to the conditional statements). Nevertheless, one should realize that the returned invariant could only be proven before our domain by a disjunctive version of at least a zone analyzer; but for `oddeven9` for instance, which consists of a sequence of 40 independent `if` blocks, a full partition into the potential $2^{40} \sim 10^{12}$ paths would have to be used to discover the same invariant, which is intractable both in time and memory.

5 Conclusion and Future Work

In this article, we described the first few applications of max-plus algebra to static analysis. Many improvements are yet to be discovered. Among these are improvements of the widening operator, to be applicable directly on a representation with generators and not on a constraint form, which would allow to compute invariants only by using the generator form for the whole analysis.

Existing memory manipulation analyzes could take advantage of our abstraction. For instance, it could be directly integrated in non-disjunctive array predicate abstractions (*e.g.* [33]), and help to automatically discover preconditions on C library functions [35] without disjunction. Moreover, when analyzing sorting algorithms, in order to prove that the resulting array is correctly ordered (and not infer information over its first and last elements only), one would need min-max-plus [36] invariants, generalizing our max-plus and min-plus invariants.

Last but not least, in order to deal with the intrinsic complexity of the full max-plus polyhedra, it is natural to think of generalizations of templates [37] to max-plus algebra. This is left for future work.

References

1. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, Springer, Heidelberg (2001)
2. Miné, A.: The octagon abstract domain. In: AST 2001 in WCRE 2001, pp. 310–319. IEEE Computer Society Press, Los Alamitos (2001)
3. Dor, N., Rodeh, M., Sagiv, M.: Cssv: towards a realistic tool for statically detecting all buffer overflows in C. In: PLDI 2003, ACM Press, New York (2003)
4. Allamigeon, X., Godard, W., Hymans, C.: Static Analysis of String Manipulations in Critical Embedded C Programs. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, Springer, Heidelberg (2006)
5. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM TOPLAS 29(5) (2007)
6. Karr, M.: Affine relationships among variables of a program. Acta Inf. 6 (1976)
7. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL 1978, Tucson, Arizona, USA. ACM Press, New York (1978)
8. Zimmermann, K.: A general separation theorem in extremal algebras. Ekonom.-Mat. Obzor. 13(2), 179–201 (1977)
9. Litvinov, G., Maslov, V., Shpiz, G.: Idempotent functional analysis: an algebraical approach. Math. Notes 69(5), 696–729 (2001); Also eprint arXiv:math.FA/0009128
10. Gaubert, S., Plus, M.: Methods and applications of (max,+) linear algebra. In: Reischuk, R., Morvan, M. (eds.) STACS 1997. LNCS, vol. 1200, Springer, Heidelberg (1997)

11. Cohen, G., Gaubert, S., Quadrat, J.P.: Max-plus algebra and system theory: where we are and where to go now. Annual Reviews in Control 23, 207–219 (1999)
12. Develin, M., Sturmfels, B.: Tropical convexity. Doc. Math. 9, 1–27 (2004)
13. Cohen, G., Gaubert, S., Quadrat, J.P.: Duality and separation theorem in idempotent semimodules. Linear Algebra and Appl. 379, 395–422 (2004)
14. Joswig, M.: Tropical halfspaces. In: Combinatorial and computational geometry. Math. Sci. Res. Inst. Publ., vol. 52. Cambridge Univ. Press, Cambridge (2005)
15. Cohen, G., Gaubert, S., Quadrat, J.P., Singer, I.: Max-plus convex sets and functions. In: Litvinov, G.L., Maslov, V.P. (eds.) Idempotent Mathematics and Mathematical Physics. Contemporary Mathematics, vol. 377, pp. 105–129. AMS (2005)
16. Gaubert, S., Katz, R.: Max-plus convex geometry. In: Schmidt, R.A. (ed.) RelMiCS/AKA 2006. LNCS, vol. 4136, pp. 192–206. Springer, Heidelberg (2006)
17. Butkovič, P., Schneider, H., Sergeev, S.: Generators, extremals and bases of max cones. Linear Algebra Appl. 421, 394–406 (2007)
18. Gaubert, S., Katz, R.: The Minkowski theorem for max-plus convex sets. Linear Algebra and Appl. 421, 356–369 (2006)
19. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977, Los Angeles, California. ACM Press, New York (1977)
20. Sankaranarayanan, S., Ivancic, F., Shlyakhter, I., Gupta, A.: Static analysis in disjunctive numerical domains. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 3–17. Springer, Heidelberg (2006)
21. Giacobazzi, R., Ranzato, F.: Compositional Optimization of Disjunctive Abstract Interpretations. In: Riis Nielson, H. (ed.) ESOP 1996. LNCS, vol. 1058, Springer, Heidelberg (1996)
22. Sotin, P., Cachera, D., Jensen, T.: Quantitative static analysis over semirings: analysing cache behaviour for java card. In: QAPL 2006. ENTCS, vol. 1380, Elsevier, Amsterdam (2006)
23. Butkovič, P., Hegedüs, G.: An elimination method for finding all solutions of the system of linear equations over an extremal algebra. Ekonomicko-matematicky Obzor. 20(2), 203–215 (1984)
24. Chernikova, N.V.: Algorithm for discovering the set of all solutions of a linear programming problem. U.S.S.R. Computational Mathematics and Mathematical Physics 8(6), 282–293 (1968)
25. Le Verge, H.: A note on Chernikova's algorithm (1992)
26. Gaubert, S., Katz, R.: External and internal representation of max-plus polyhedra. Privately circuled draft (2008)
27. Baccelli, F., Cohen, G., Olsder, G.J., Quadrat, J.P.: Synchronization and Linearity. Wiley, Chichester (1992)
28. Cohen, G., Gaubert, S., Quadrat, J.P.: Regular matrices in max-plus algebra (preprint, 2008)
29. Halbwachs, N.: Détermination Automatique de Relations Linéaires Vérifiées par les Variables d'un Programme. Thèse de $3^{\text{ème}}$ cycle d'informatique, Université scientifique et médicale de Grenoble, Grenoble, France (March 1979)
30. Halbwachs, N., Proy, Y., Roumanoff, P.: Verification of real-time systems using linear relation analysis. Formal Methods in System Design 11(2) (August 1997)
31. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 348–363. Springer, Heidelberg (2005)
32. Gopan, D., Reps, T., Sagiv, M.: A framework for numeric analysis of array operations. SIGPLAN Not. 40(1) (2005)

33. Allamigeon, X.: Non-disjunctive Numerical Domain for Array Predicate Abstraction. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 163–177. Springer, Heidelberg (2008)
34. Batcher, K.: Sorting networks and their applications. In: Proceedings of the AFIPS Spring Joint Computer Conference 32, pp. 307–314 (1968)
35. Moy, Y.: Sufficient preconditions for modular assertion checking. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905. Springer, Heidelberg (2008)
36. Gaubert, S., Gunawardena, J.: The duality theorem for min-max functions. C. R. Acad. Sci. Paris. 326 (Série I), 43–48 (1998)
37. Sankaranarayanan, S., Sipma, H., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, Springer, Heidelberg (2005)

Conflict Analysis of Programs with Procedures, Dynamic Thread Creation, and Monitors

Peter Lammich and Markus Müller-Olm

Institut für Informatik, Fachbereich Mathematik und Informatik
Westfälische Wilhelms-Universität Münster
`peter.lammich@uni-muenster.de`, `mmo@math.uni-muenster.de`

Abstract. We study conflict detection for programs with procedures, dynamic thread creation and a fixed finite set of (reentrant) monitors. We show that deciding the existence of a conflict is NP-complete for our model (that abstracts guarded branching by nondeterministic choice) and present a fixpoint-based complete conflict detection algorithm. Our algorithm needs worst-case exponential time in the number of monitors, but is linear in the program size.

1 Introduction

As programming languages with explicit support for parallelism, such as Java, have become popular, the interest in analysis of parallel programs has increased in recent years. A particular problem of parallel programs are conflict situations, where a program is simultaneously in two states that should exclude each other. An example is a data race, where a memory location is simultaneously accessed by two threads, at least one of the accesses being a write access. Particular challenges for analyzing conflicts in a language like Java are dynamic creation of an unlimited number of threads, synchronization via reentrant monitors, and indirect referencing of monitors by their association to objects. It's unrealistic to design an interprocedural analysis that meets all these challenges and is precise. Therefore, this paper concentrates on the first two challenges. More specifically, we develop an analysis that decides the reachability of a conflict situation in (nondeterministic) programs with recursive procedure calls, dynamic thread creation and synchronization via a fixed finite set of reentrant monitors that are statically bound to procedures. We also show this problem to be NP-complete.

Many papers on precise program analysis, e.g. [4,13,16], model concurrency via parbegin/parend blocks or parallel procedure calls or assume a fixed set of threads. However, thread-creation cannot be simulated by parbegin/parend for programs with procedures [2]. Precise analysis for programs with thread-creation is treated e.g. in [2,11]. All the papers mentioned above completely abstract away synchronization. Due to a well-known result of Ramalingam [15], context- and synchronization-sensitive analysis is undecidable. However, Ramalingam considered rendezvous style synchronization which is more powerful than synchronization via monitors studied in this paper. The undecidability border

© Springer-Verlag Berlin Heidelberg 2008

for various properties and synchronization primitives in models with a fixed set of threads is studied more closely in [7]. An interesting approach to address the indirect referencing of monitors extends shape analysis techniques to a concurrent scenario (e.g. [17]). However, this approach does not come with a precision theorem but assesses precision empirically.

The work that comes closest to our goals is by Kahlon et al. [8,6]. They study analyses for a fixed set of threads communicating via (statically referenced) locks, where each thread is modeled by a pushdown system (which corresponds to programs with recursive procedures). In their model, threads can execute lock/unlock statements for a fixed finite set of locks in a nested fashion, i.e. each thread can only release the lock it acquired last and that was not yet released. Without this nestedness constraint data race detection is undecidable [8,7]. Moreover, locks are not reentrant in their model, i.e. a thread may not reacquire a lock that it possesses already[1]. In contrast, our model uses reentrant monitors (i.e. ,,synchronized"-blocks). Monitors correspond to a structured use of nested locks. A monitor can be interpreted as a lock that is acquired upon entering a synchronized block and released when leaving the synchronized block. While synchronized blocks start and end in the same procedure, the lock and unlock statements in [8,6] can occur anywhere in the program. However, since languages like Java also use synchronized blocks, we believe this restriction being rather harmless.

Our contributions beyond the work of [8] are as follows: We handle thread creation instead of assuming a fixed set of threads; we handle reentrant monitors instead of non-reentrant nested locks; and we use fixpoint based instead of automata based techniques. The running time of our algorithm depends only linearly from the program size, independently of the number of threads actually created. In contrast, the running time of the algorithm of [8] grows at least quadratically with the (statically fixed) number of threads. Moreover, we show conflict analysis to be NP-complete for our model, where NP-hardness also transfers to models with a fixed set of threads including the one of Kahlon et al., which justifies the exponential dependency of the running time of the analysis algorithms from the number of monitors.

This paper is organized as follows: In Section 2 the program model and its semantics is defined. In Section 3, we define an alternative operational semantics that captures only particular schedules, called *restricted executions*. We show that any reachable configuration is also reachable by a restricted execution. In Section 4, we characterize restricted executions by a constraint system. In Section 5, we interpret this constraint system over an abstract domain, obtaining a fixpoint based conflict analysis algorithm needing exponential time in the number of monitors and linear time in the program size. As part of the abstract domain, we use the concept of acquisition histories [8], adapted to reentrant monitors and to a fixpoint based analysis context. Additionally, we show that the conflict analysis problem is NP-complete . Finally, in Section 6, we give a

[1] One can simulate reentrant locks by non-reentrant ones, but at the cost of a worst-case exponential blowup of the program size

conclusion and an outlook to further research. Due to the lack of space, most of the proofs are deferred to an accompanying technical report [9].

2 Program Model

Flowgraphs. We describe programs by nondeterministic interprocedural flowgraphs with monitors. A program $\Pi = (\mathsf{P}, \mathcal{M}, (G_p, \mathsf{m}(p))_{p\in\mathsf{P}})$ consists of a finite set P of procedure names with $\mathsf{main} \in \mathsf{P}$ and a finite set $\mathcal{M}$ of monitor names. Each procedure $p \in \mathsf{P}$, is associated with a flowgraph G_p, describing the body of the procedure, and a set of monitors $\mathsf{m}(p) \subseteq \mathcal{M}$ describing the monitors the procedure synchronizes on, i.e. the monitors in $\mathsf{m}(p)$ are acquired upon entering p and released upon returning from p. A flowgraph $G_p = (\mathsf{N}_p, \mathsf{E}_p, \mathsf{e}_p, \mathsf{r}_p)$ consists of a finite set N_p of control nodes, a set $\mathsf{E}_p \subseteq \mathsf{N}_p \times \mathcal{L}_{\mathsf{Edge}} \times \mathsf{N}_p$ of labeled edges, and distinguished entry and return nodes $\mathsf{e}_p, \mathsf{r}_p \in \mathsf{N}_p$. Edges are labeled with either base, call, or spawn labels: $\mathcal{L}_{\mathsf{Edge}} = \{\mathsf{base}\} \cup \{\mathsf{call}\ p \mid p \in \mathsf{P}\} \cup \{\mathsf{spawn}\ p \mid p \in \mathsf{P}\}$. Intuitively, base edges model basic instructions. We leave there structure unspecified as we will define conflicting situations on the basis of the control state of the program rather than by the executed instructions. Call edges model (potentially recursive) procedure calls and spawn edges model thread creation. As usual, we assume $\mathsf{N}_p \cap \mathsf{N}_{p'} = \emptyset$ for $p \neq p' \in \mathsf{P}$ and define $\mathsf{N} := \bigcup_{p\in\mathsf{P}} \mathsf{N}_p$ and $\mathsf{E} := \bigcup_{p\in\mathsf{P}} \mathsf{E}_p$. We call a procedure $p \in \mathsf{P}$ *initial*, if it is the main procedure ($p = \mathsf{main}$) or a procedure started by a spawn edge ($\exists u, v.(u, \mathsf{spawn}\ p, v) \in \mathsf{E}$). In order to avoid the (unrealistic) possibility that a thread can allocate monitors in the moment it is created, we assume that initial procedures do not synchronize on any monitors. Otherwise, the analysis problem would be more complex (PSPACE-hard). For the rest of this paper, we assume a fixed program $\Pi = (\mathsf{P}, \mathcal{M}, (G_p, \mathsf{m}(p))_{p\in\mathsf{P}})$. In order to simplify the definition of restricted executions in Section 3, we agree on some further conventions: For each initial procedure p we have $\mathsf{e}_p \neq \mathsf{r}_p$ and there is a procedure q such that $\mathsf{E}_p = \{(\mathsf{e}_p, \mathsf{call}\ q, \mathsf{r}_p)\}$, $\mathsf{e}_q \neq \mathsf{r}_q$, and $\mathsf{E}_q \cap \{(u, l, \mathsf{r}_q) \mid u \in \mathsf{N}_q \wedge l \in \mathcal{L}_{\mathsf{Edge}}\} = \emptyset$, i.e. the return node of q is isolated. These syntactic restrictions guarantee that the execution of any thread starts with a call to a procedure that will never return. However, they do not limit the expressiveness of our model, since we can always rewrite a program to one with the same set of conflicts that satisfies these restrictions, e.g. by introducing starter procedures x_0 and x_1 for every initial procedure x as illustrated in Fig. 1 and replacing $\mathsf{spawn}\ x$ by $\mathsf{spawn}\ x_0$, as well as main by main_0. Note that reaching control node t_x models termination of a thread: Arrived there, it holds no monitors and cannot make any steps.

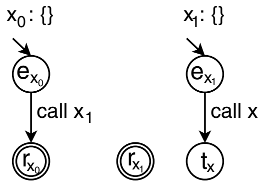


Fig. 1. Starter procedure

The flowgraph depicted in Fig. 2 is used as a running example throughout this paper. Its main procedure is a_0. For better readability, the x_0 and x_1 procedures (for $x \in \{a, p, r\}$) as well as the base edge labels are not shown. First of all, let us illustrate some effects our analysis has to cope with. Consider nodes 5 and 9. In order to reach 5, we have to call p and then pass through t and in order to reach 9, we have to call q and then pass through s. If one thread calls p first, it is in monitor m_1, and no other thread can pass through s any more. Vice versa, if q is called first, no other thread can pass through t any more. Hence nodes 5 and 9 are not simultaneously reachable, although the sets of monitors held at 5, $\{m_1\}$, and 9, $\{m_2\}$, are disjoint. We will use the concept of *acquisition histories* [8] to handle this effect. Now consider nodes 3 and C. They are simultaneously reachable, because procedure q can create a thread starting with r_0, and after q has returned to node 3, the new thread can pass through procedure t and reach node C. This illustrates that a thread survives the procedure it is created in. In order to cope with this, we need complex procedure summaries. Finally, consider nodes 8 and C. They are not simultaneously reachable, as the thread starting at r_0 cannot pass through procedure t until the initial thread has released monitor m_2. However, if we had no thread creation, but two initial threads of type main and r, the nodes 8 and C would be simultaneously reachable. This illustrates that conflicts in our model with thread creation cannot (easily) be reduced to models with a fixed set of initial threads (as covered by Kahlon et al. [8,6]).

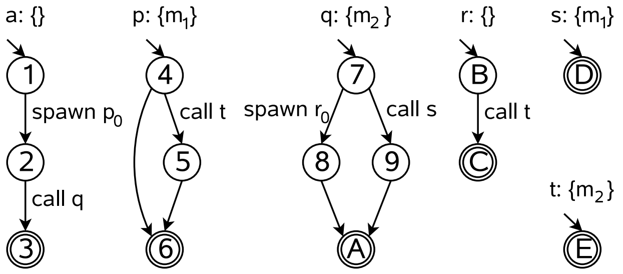


Fig. 2. Example flowgraph with $\mathsf{main} = a_0$

Operational Semantics. We use the following multiset and list notations: $\mathsf{mset}(R)$ is the set of multisets of elements from R, $\emptyset$ is the empty multiset, $\{x\}$ is the multiset containing x once and $R_1 \uplus R_2$ is the union of the multisets R_1 and R_2. The ambiguity between set and multiset notation is resolved from the context. R^* is the set of lists of elements from R, ε is the empty list, $[e_1, \ldots, e_k]$ is the list of elements $e_1, \ldots, e_k$, and $w_1 w_2$ is the concatenation of the lists w_1 and w_2.

The operational semantics is described as a labeled transition system $\overset{\cdot}{\longrightarrow} \subseteq \mathsf{Conf} \times \mathcal{L}_{\mathsf{LE}} \times \mathsf{Conf}$, where $\mathsf{Conf} := \{\langle s, c\rangle \mid s \in \mathsf{N}^*, c \in \mathsf{mset}(\mathsf{N}^*), \mathsf{cons}(\{s\} \uplus c)\}$ is the set of *monitor consistent* program configurations and $\mathcal{L}_{\mathsf{LE}} := \{l^x | l \in \mathcal{L} \wedge x \in \{\mathsf{L}, \mathsf{E}\}\}$ with $\mathcal{L} := \mathcal{L}_{\mathsf{Edge}} \cup \{\mathsf{ret}\}$ is the set of transition labels. A program configuration $\langle s, c\rangle \in \mathsf{Conf}$ is a pair of a local thread's configuration s and a multiset c of environment threads' configurations. A thread's configuration is modeled as a stack of control nodes, the top element being the current control node and the elements deeper in the stack being stored return addresses. We model stacks as lists with the top of the stack being the first element of the list. For a control node $u \in \mathsf{N}_p$, we define $\mathsf{m}(u) := \mathsf{m}(p)$. For a stack $s \in \mathsf{N}^*$ we define $\mathsf{m}(s) := \bigcup_{n\in s}\mathsf{m}(n)$, for $c \in \mathsf{mset}(\mathsf{N}^*)$, we define $\mathsf{m}(c) = \bigcup_{s\in c}\mathsf{m}(s)$ and for $\langle s, c\rangle \in \mathsf{Conf}$ we define $\mathsf{m}(\langle s, c\rangle) := \mathsf{m}(\{s\} \uplus c)$. A multiset of stacks $c \in \mathsf{mset}(\mathsf{N}^*)$ is *monitor consistent*, if no two threads are inside the same monitor. This is expressed by the predicate $\mathsf{cons}(c) :\Leftrightarrow \nexists s_1, s_2, c_e.\ c = \{s_1\} \uplus \{s_2\} \uplus c_e \wedge \mathsf{m}(s_1) \cap \mathsf{m}(s_2) \neq \emptyset$. Transitions are labeled with the edge that induced the transition or with the ret label for a procedure return. Additionally, we record whether the transition was made on the local thread ($\cdot^{\mathsf{L}}$) or on some environment thread ($\cdot^{\mathsf{E}}$). This distinction is needed when characterizing certain sets of executions by a constraint system in order to distinguish the monitors used by the environment threads from the monitors used by the local thread. We define $\xrightarrow{\cdot}$ to be the least set satisfying the following rules:

$$\begin{array}{llll}
[\mathsf{base}] & (u, \mathsf{base}, v) \in \mathsf{E}: & \langle [u]r, c\rangle \xrightarrow{\mathsf{base}^{\mathsf{L}}} \langle [v]r, c\rangle & \\
[\mathsf{call}] & (u, \mathsf{call}\ q, v) \in \mathsf{E}: & \langle [u]r, c\rangle \xrightarrow{(\mathsf{call}\ q)^{\mathsf{L}}} \langle [\mathsf{e}_q, v]r, c\rangle & \text{if } \mathsf{m}(q) \cap \mathsf{m}(c) = \emptyset \\
[\mathsf{ret}] & q \in \mathsf{P}: & \langle [\mathsf{r}_q]r, c\rangle \xrightarrow{\mathsf{ret}^{\mathsf{L}}} \langle r, c\rangle & \\
[\mathsf{spawn}] & (u, \mathsf{spawn}\ q, v) \in \mathsf{E}: & \langle [u]r, c\rangle \xrightarrow{(\mathsf{spawn}\ q)^{\mathsf{L}}} \langle [v]r, \{[\mathsf{e}_q]\} \uplus c\rangle & \\
[\mathsf{env}] & & \langle s, \{r\} \uplus c\rangle \xrightarrow{l^{\mathsf{E}}} \langle s, \{r'\} \uplus c'\rangle \text{ if } \langle r, \{s\} \uplus c\rangle \xrightarrow{l^{\mathsf{L}}} \langle r', \{s\} \uplus c'\rangle &
\end{array}$$

The [base], [call], and [spawn]-rules model the behavior of the corresponding edges. Returning from procedures is modeled by the [ret]-rule. Note that there is no flowgraph edge corresponding to a return step. Finally, the [env]-rule defines the environment steps.

We overload $\xrightarrow{\cdot}$ with its reflexive transitive closure ($\langle s, c\rangle \xrightarrow{w} \langle s', c'\rangle$ with $w \in \mathcal{L}_{\mathsf{LE}}{}^*$) and write $\xrightarrow{*}$ for the execution of an arbitrary path. For $x \in \{\mathsf{L}, \mathsf{E}\}$ and $w = [l_1, \ldots, l_n] \in \mathcal{L}^*$ we define $w^x := [l_1^x, \ldots, l_n^x]$ and write $c \xrightarrow{w} c'$ as a shorthand notation for $\langle \varepsilon, c\rangle \xrightarrow{w^{\mathsf{E}}} \langle \varepsilon, c'\rangle$. As the empty stack cannot make any steps and holds no monitors, it does not influence the execution. Thus $c \xrightarrow{l} c'$ simply is a transition without explicit local thread. Our semantics preserves monitor consistency of the configurations as the monitor side condition in the [call]-rule ensures that a thread can only enter a procedure if no other thread is inside a monitor the procedure synchronizes on.

The monitors used by a path are the monitors of all procedures that are called on steps of this path: For $l \in \mathcal{L}$, we define $\mathsf{m}(l) := \mathsf{m}(p)$ if $l = \mathsf{call}\ p$ and $\mathsf{m}(l) = \emptyset$ otherwise. We overload this definition for sequences of labels ($\mathsf{m}(w) := \bigcup_{l\in w}\mathsf{m}(l)$ for $w \in \mathcal{L}^*$). For sequences with L/E-labeling $w \in \mathcal{L}_{\mathsf{LE}}{}^*$, we define $\mathsf{m}^{\mathsf{L}}(w)$ to be the set of monitors used by local steps and $\mathsf{m}^{\mathsf{E}}(w)$ to be the set of monitors used by environment steps.

Reachability of a Conflict. For a multiset $C = \{U_1, \ldots, U_n\} \in \mathsf{mset}(2^{\mathsf{N}})$ of sets of nodes and a multiset of stacks $c \in \mathsf{mset}(\mathsf{N}^*)$, we define $\mathsf{at}_C(c)$ if and only if $c = c_e \uplus \biguplus_{i=1\ldots n}\{[u_i]r_i\}$ for some $c_e \in \mathsf{mset}(\mathsf{N}^*)$, $(r_i \in \mathsf{N}^*)_{i=1\ldots n}$, and $(u_i \in U_i)_{i=1\ldots n}$, i.e. for each i, c contains an own thread with current control node in U_i. We also define $\mathsf{at}_C(\langle s, c\rangle) :\Leftrightarrow \mathsf{at}_C(\{s\} \uplus c)$. The conflict analysis problem for two sets of control nodes $U, V \subseteq \mathsf{N}$ is to decide the question: *Is there an execution* $\{[\mathsf{e_{main}}]\}\xrightarrow{*}c$ *with* $\mathsf{at}_{\{U,V\}}(c)$? The reachability problem for a single set U is to decide: *Is there an execution* $\{[\mathsf{e_{main}}]\}\xrightarrow{*}c$ *with* $\mathsf{at}_{\{U\}}(c)$?

Example 1. In the following example executions, we abbreviate $\mathsf{call}\ p$ by p and $\mathsf{spawn}\ p$ by $+p$. Fig. 3a illustrates an execution of the flowgraph from Fig. 2. The execution starts with a single thread at the entry point of a_0. It calls procedures a_1, a, and then passes through procedure q. On its way, it spawns two other threads p_0 and r_0. Their steps are interleaved with the initial thread's ones. This execution can be formally described as $\langle \mathsf{e}_{a_0}, \emptyset\rangle \xrightarrow{w} c'$ with transition labels $w = [{a_1}^{\mathsf{L}}, a^{\mathsf{L}}, +{p_0}^{\mathsf{L}}, q^{\mathsf{L}}, {p_1}^{\mathsf{E}}, +{r_0}^{\mathsf{L}}, {r_1}^{\mathsf{E}}, r^{\mathsf{E}}, \mathsf{base}^{\mathsf{L}}, \mathsf{ret}^{\mathsf{L}}, t^{\mathsf{E}}, p^{\mathsf{E}}, \mathsf{ret}^{\mathsf{E}}]$ and end configuration $c' = \langle [3, t_a, \mathsf{r}_{a_0}], \{[C, t_r, \mathsf{r}_{r_0}], [4, t_p, \mathsf{r}_{p_0}]\}\rangle$. Note that this execution witnesses the conflict between nodes 3 and C mentioned above.

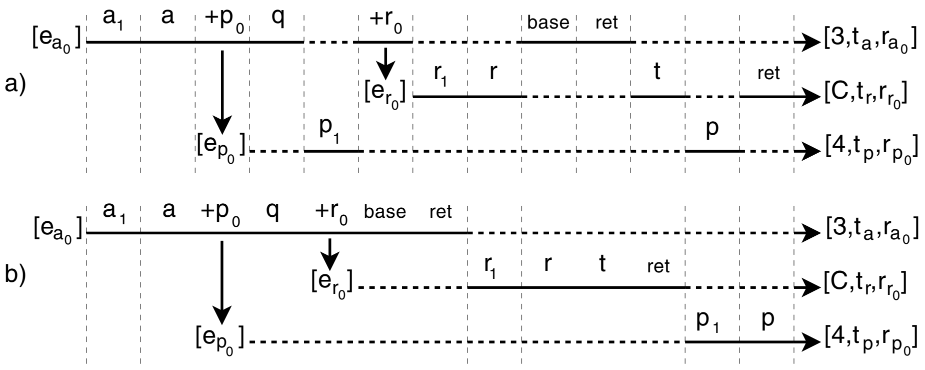


Fig. 3. Sample execution and corresponding restricted execution

3 Restricted Schedules

In this section we define a restricted operational semantics that only allows a subset of the executions of the original semantics, but preserves the set of reachable configurations, and thus the reachable conflicts. The restricted semantics is better suited for characterization by a constraint system than the original semantics (cf. Section 4).

While in an execution of the original semantics, context switches may occur after each step, the restricted semantics only allows context switches after a thread's last step and before procedure calls that do not return for the rest of the execution. Due to the syntactic convention that assures that the execution of any thread starts with a non-returning call, an atomically scheduled sequence, called a *macrostep*, consists of an initial procedure call, followed by a *same-level path*. A same-level path is a path with balanced calls and returns, i.e. its execution starts and ends at the same stack level, and does not fall below the initial stack level at any point. We define the transition relation of the restricted semantics $\overset{\cdot}{\Longrightarrow} \subseteq \mathsf{Conf} \times \mathsf{MStep} \times \mathsf{Conf}$ with $\mathsf{MStep} := \{([\mathsf{call}\ p]\bar{w})^x \mid p \in \mathsf{P}, \bar{w} \in \mathcal{L}^*, x \in \{\mathsf{L}, \mathsf{E}\}\}$ as the least set satisfying the following rules:

$$\begin{array}{lll} [\mathsf{macro}] & \langle s,c\rangle \overset{([\mathsf{call}\ p]\bar{w})^{\mathsf{L}}}{\Longrightarrow} \langle [v]r',c'\rangle & \text{if } \langle s,c\rangle \xrightarrow{(\mathsf{call}\ p)^{\mathsf{L}}} \langle [\mathsf{e}_p]r',c\rangle \wedge \langle [\mathsf{e}_p],c\rangle \xrightarrow{\bar{w}^{\mathsf{L}}} \langle [v],c'\rangle \\ [\mathsf{env}] & \langle s,\{r\} \uplus c\rangle \overset{l^{\mathsf{E}}}{\Longrightarrow} \langle s,\{r'\} \uplus c'\rangle & \text{if } \langle r,\{s\} \uplus c\rangle \overset{l^{\mathsf{L}}}{\Longrightarrow} \langle r',\{s\} \uplus c'\rangle \end{array}$$

The [macro]-rule captures the intuition of a macrostep of the local thread as described above and the [env]-rule infers the environment steps. Note that a same-level execution is written as $\langle [u],c\rangle \xrightarrow{\bar{w}^{\mathsf{L}}} \langle [v],c'\rangle$. As monitors are reentrant, it also implies the executions $\langle [u]r',c\rangle \xrightarrow{\bar{w}^{\mathsf{L}}} \langle [v]r',c'\rangle$ for any stack r', s.t. $\langle [u]r',c\rangle$ is monitor consistent. We extend $\overset{\cdot}{\Longrightarrow}$ to its reflexive transitive closure, write $\overset{*}{\Longrightarrow}$ for the execution of an arbitrary path, and define $c \overset{w}{\Longrightarrow} c' := \langle \varepsilon, c\rangle \overset{w^{\mathsf{E}}}{\Longrightarrow} \langle \varepsilon, c'\rangle$. Note that a transition is then labeled by a sequence of macrosteps $w = [l_1^{x_1}, \ldots, l_n^{x_n}] \in \mathsf{MStep}^*$ with $x_1, \ldots, x_n \in \{\mathsf{L}, \mathsf{E}\}$ where each macrostep $l_i \in \mathsf{MStep}$ $(1 \leq i \leq n)$ is labeled as either a local (l_i^{L}) or an environment (l_i^{E}) step.

As macrosteps always start with a call that does not return for the rest of the execution, the set of allocated monitors does not decrease during an execution:

Theorem 2. *An execution $\langle s,c\rangle \overset{w}{\Longrightarrow} \langle s',c'\rangle$ implies $\mathsf{m}(\langle s,c\rangle) \subseteq \mathsf{m}(\langle s',c'\rangle)$.*

Starting with an initial procedure, the sets of configurations reachable by executions of the original semantics and of the restricted semantics are the same:

Theorem 3. *Let $p \in \mathsf{P}$ be an initial procedure. A configuration can be reached from p by an execution of the original semantics if and only if it can be reached by an execution of the restricted semantics. Formally: $\{[\mathsf{e}_p]\} \xrightarrow{*} c' \Leftrightarrow \{[\mathsf{e}_p]\} \overset{*}{\Longrightarrow} c'$.*

Example 4. Fig. 3b illustrates a restricted execution that reaches the same configuration c' as the execution from Fig. 3a. It is described as $\langle \mathsf{e}_{a_0}, \emptyset\rangle \overset{w'}{\Longrightarrow} c'$ with $w' = [[a_1]^{\mathsf{L}}, [a, +p_0, q, +r_0, \mathsf{base}, \mathsf{ret}]^{\mathsf{L}}, [r_1]^{\mathsf{E}}, [r, t, \mathsf{ret}]^{\mathsf{E}}, [p_1]^{\mathsf{E}}, [p]^{\mathsf{E}}]$.

Monitor Consistent Interleaving. For a single macrostep $l = [\mathsf{call}\ p]\bar{w}$, we define $\mathsf{ent}(l) := \mathsf{m}(p)$ and $\mathsf{pass}(l) = \mathsf{m}(\bar{w})$, i.e. the monitors that are entered and never exited by a macrostep and the monitors that are passed (entered and exited again), respectively. We inductively define the monitor consistent interleaving operator $\otimes : \mathsf{MStep}^* \times \mathsf{MStep}^* \to \mathsf{MStep}^*$ by $\varepsilon \otimes w = w \otimes \varepsilon = \{w\}$ and $[l_1]w_1 \otimes [l_2]w_2 := \bigcup_{i=1,2}\{[l_i]w \mid w \in w_i \otimes [l_{3-i}]w_{3-i} \wedge \mathsf{ent}(l_i) \cap \mathsf{m}([l_{3-i}w_{3-i}]) = \emptyset\}$. Note that the $\otimes$-operator is not aware of the L/E-labeling of its operands, it just copies the labeling to the result. Monitor consistent interleaving is a restriction of the usual interleaving to those interleavings where no monitor is used by one path if it has been entered by the other path. For example, in the flowgraph of Fig. 2, we have $[[r, t, \mathsf{ret}]^{\mathsf{E}}] \otimes [[q]^{\mathsf{L}}] = \{[[r, t, \mathsf{ret}]^{\mathsf{E}}, [q]^{\mathsf{L}}]\}$. Note that the macrostep sequence $[[q]^{\mathsf{L}}, [r, t, \mathsf{ret}]^{\mathsf{E}}]$ is not a monitor consistent interleaving, as q enters monitor m_2 that inhibits execution of the macrostep $[r, t, \mathsf{ret}]^{\mathsf{E}}$. We show that the $\otimes$-operator captures the behavior of our interleaving semantics:

Theorem 5. *For configurations $\langle s, c_1 \uplus c_2\rangle$, $\langle s', c'\rangle \in \mathsf{Conf}$ and a macrostep path $w \in \mathsf{MStep}^*$, we have $\langle s, c_1 \uplus c_2\rangle \overset{w}{\Longrightarrow} \langle s', c'\rangle$ if and only if there exist $w_1, w_2 \in \mathsf{MStep}^*$ with $w \in w_1 \otimes w_2^{\mathsf{E}}$ and $c'_1, c'_2 \in \mathsf{mset}(\mathsf{N}^*)$ with $c' = c'_1 \uplus c'_2$, $\langle s, c_1\rangle \overset{w_1}{\Longrightarrow} \langle s', c'_1\rangle$, $c_2 \overset{w_2}{\Longrightarrow} c'_2$, $\mathsf{m}(\langle s, c_1\rangle) \cap \mathsf{m}(c_2) = \emptyset$, $\mathsf{m}(\langle s, c_1\rangle) \cap \mathsf{m}(w_2) = \emptyset$, and $\mathsf{m}(c_2) \cap \mathsf{m}(w_1) = \emptyset$.*

Intuitively, this theorem states that we can split an execution by the threads in its starting configuration into interleavable executions, and, vice versa, can combine interleavable executions into one execution. Note that the interleaving operator $\otimes$ only ensures that monitors allocated by one execution do not interfere with the monitors used by the other execution, but is not aware of the monitors of the start configurations. Hence, the last three conditions in this theorem ensure that the resulting combined configuration is monitor consistent and that the monitors of the start configuration of one execution do not interfere with the monitors used by the other execution.

Example 6. Consider the executions $\langle [7], \emptyset\rangle \overset{[s]^{\mathsf{L}}}{\Longrightarrow} \langle [D, 9], \emptyset\rangle$ and $\{[4]\} \overset{[t]^{\mathsf{E}}}{\Longrightarrow} \{[E, 5]\}$ of the flowgraph in Fig. 2. Although $w := [[s]^{\mathsf{L}}, [t]^{\mathsf{E}}] \in [[s]^{\mathsf{L}}] \otimes [[t]^{\mathsf{E}}]$, there is no execution $\langle [7], [4]\rangle \overset{w}{\Longrightarrow} \langle [D, 9], \{[E, 5]\}\rangle$, as $\mathsf{m}(\{4\}) \cap \mathsf{m}([[s]^{\mathsf{L}}]) = \{m_1\} \neq \emptyset$.

4 Constraint Systems

In this section we develop a constraint system based characterization of restricted executions starting at a single control node. For a procedure $p \in \mathsf{P}$, we want to represent all executions starting with a call of p and reaching some configuration $\langle s, c\rangle$. However, we omit the initial procedure call in order to make the execution independent from the monitors held at the call site. The representation of such an execution, called a *reaching triple*, is a triple of the first macrostep's same-level path, the remaining macrosteps, and the reached configuration:

$$R^{\mathsf{op}}[p] := \{(\bar{w}, w, \langle s, c\rangle) \mid \exists \tilde{u}, \tilde{c}.\ \langle [\mathsf{e}_p], \emptyset\rangle \xrightarrow{\bar{w}^{\mathsf{L}}} \langle [\tilde{u}], \tilde{c}\rangle \xrightarrow{w} \langle s, c\rangle\}$$

The procedure summary information $S^{\mathsf{op}}[u]$ is collected for each control node u in a forwards manner; thus the actual summary for procedure p is $S^{\mathsf{op}}[\mathsf{r}_p]$. It contains triples of a same-level path $\bar{w}$ from the procedure's entry node to u, a macrostep path w of the threads spawned during the execution of the same-level path and the configuration c reached by those threads: $S^{\mathsf{op}}[u] := \{(\bar{w}, w, \langle \varepsilon, c\rangle) \mid \exists \tilde{c}.\ \langle [\mathsf{e}_p], \emptyset\rangle \xrightarrow{\bar{w}^{\mathsf{L}}} \langle [u], \tilde{c}\rangle \wedge \langle \varepsilon, \tilde{c}\rangle \xrightarrow{w} \langle \varepsilon, c\rangle\}$. Note that an artificial ε-component is included in the entries of $S^{\mathsf{op}}[u]$ in order to have entries of the same form in S^{op} and R^{op}. Thus we have $R^{\mathsf{op}}[p], S^{\mathsf{op}}[u] \subseteq \mathsf{D}$ for $\mathsf{D} := \mathcal{L}^* \times \mathsf{MStep}^* \times \mathsf{Conf}$. This allows us to handle these sets more uniformly. The last two elements of a procedure-summary triple collect information about steps that may be executed after the procedure has returned. This accounts for the fact that spawned threads survive the procedure they where created in. Note how restricted schedules reduce the complexity here: If we would collect arbitrarily scheduled executions, we would have to interleave a prefix of the steps of the created threads with the same-level path and record the suffix in the second component. This would complicate the constraint system and also the monitor consistent interleaving operator.

Both, the reaching and the same-level information have a closure property, that results from the fact that the empty path is always executable. Formally, $R^{\mathsf{op}}[p], S^{\mathsf{op}}[u] \in \mathsf{L}_R \subseteq 2^{\mathsf{D}}$ with $\mathsf{L}_R := \{X \mid \forall (\bar{w}, w, \langle s, c\rangle) \in X.\ \exists \tilde{s}, \tilde{c}.\ (\bar{w}, \varepsilon, \langle \tilde{s}, \tilde{c}\rangle) \in X\}$. This closure property is important for the abstraction done later, as it allows us to ignore steps that are not necessary to reach the conflict. Moreover, we have $S^{\mathsf{op}}[u] \in \mathsf{L}_S \subseteq \mathsf{L}_R$ for $\mathsf{L}_S := \{X \in \mathsf{L}_R \mid \forall (\bar{w}, w, \langle s, c\rangle) \in X.\ s = \varepsilon\}$. Both L_R and L_S ordered by set inclusion are complete sub-lattices of $(2^{\mathsf{D}}, \subseteq)$.

Example 7. The executions $\langle [7], \emptyset\rangle \xrightarrow{\bar{w}_1^{\mathsf{L}}} \langle [9], \emptyset\rangle$, $\langle [1], \emptyset\rangle \xrightarrow{+p_0{}^{\mathsf{L}}} \langle [2], \{[\mathsf{e}_{p_0}]\}\rangle$, and $\{[\mathsf{e}_{p_0}]\} \xRightarrow{w_2} \{[5, t_p, \mathsf{r}_{p_0}]\}$ with $\bar{w}_1 := [s, \mathsf{ret}]$ and $w_2 := [[p_1]^{\mathsf{E}}, [p, t, \mathsf{ret}]^{\mathsf{E}}]$ of the flowgraph from Fig. 2 give rise to the reaching triples $\mathsf{Tr}_1 := (\bar{w}_1, \varepsilon, \langle [9], \emptyset\rangle) \in R^{\mathsf{op}}[q]$ and $\mathsf{Tr}_2 := ([+p_0], w_2, \langle \varepsilon, \{[5, t_p, \mathsf{r}_{p_0}]\}\rangle) \in S^{\mathsf{op}}[2]$. Moreover, as $\{[\mathsf{e}_{p_0}]\} \xRightarrow{\varepsilon} \{[\mathsf{e}_{p_0}]\}$ is also a valid execution, we have $([+p_0], \varepsilon, \langle \varepsilon, \{[\mathsf{e}_{p_0}]\}\rangle) \in S^{\mathsf{op}}[2]$, witnessing the closure property.

We characterize R^{op} and S^{op} as the least solution of the following system of inequations (constraint system), where the variables $(R[p])_{p\in\mathsf{P}}$ range over L_R and $(S[u])_{u\in\mathsf{N}}$ range over L_S.

$$
\begin{array}{lll}
[\mathsf{REMPTY}]\ u \in \mathsf{N}_p : & R[p] & \supseteq S[u]|_{\neg\mathsf{m}(p)} * \{(\varepsilon, \varepsilon, \langle [u], \emptyset\rangle)\} \\
[\mathsf{RCALL}]\ (u, \mathsf{call}\ q, v) \in \mathsf{E}_p : & R[p] & \supseteq S[u]|_{\neg\mathsf{m}(p)} * ((u, \mathsf{call}\ q, v); R[q]) \\
[\mathsf{SEMPTY}]\ p \in \mathsf{P} : & S[\mathsf{e}_p] & \supseteq \{\varepsilon, \varepsilon, \langle \varepsilon, \emptyset\rangle\} \\
[\mathsf{SBASE}]\ (u, \mathsf{base}, v) \in \mathsf{E}_p : & S[v] & \supseteq S[u] * \mathsf{base} \\
[\mathsf{SCALL}]\ (u, \mathsf{call}\ q, v) \in \mathsf{E}_p : & S[v] & \supseteq S[u] * \mathsf{call}\ q * S[\mathsf{r}_q] * \mathsf{ret} \\
[\mathsf{SSPAWN}]\ (u, \mathsf{spawn}\ q, v) \in \mathsf{E}_p : & S[v] & \supseteq S[u] * \mathsf{spawn}\ q * \mathsf{env}(R[q])
\end{array}
$$

Here l is an abbreviation of $\{([l], \varepsilon, \langle \varepsilon, \emptyset\rangle)\}$ for $l \in \{\mathsf{base}, \mathsf{call}\ q, \mathsf{ret}, \mathsf{spawn}\ q\}$. The operator $\cdot|_{\neg M} : \mathsf{L}_S \to \mathsf{L}_S$ is defined by $X|_{\neg M} := X \cap \{(\bar{w}, w, \langle s, c\rangle) \mid \mathsf{m}^{\mathsf{E}}(w) \cap M = \emptyset\}$. The operators $(u, \mathsf{call}\ q, v); \cdot : \mathsf{L}_R \to \mathsf{L}_R$, $\mathsf{env}(\cdot) : \mathsf{L}_R \to \mathsf{L}_S$, and $* : \mathsf{L}_S \times \mathsf{L}_R \to \mathsf{L}_R$ are the natural extensions to sets of the following definitions:

$$
\begin{array}{ll}
(u, \mathsf{call}\ q, v); (\bar{w}, w, \langle s, c\rangle) & := \{(\varepsilon, \varepsilon, \langle [u], \emptyset\rangle)\} \cup \\
& \quad \{(\varepsilon, [([\mathsf{call}\ q]\bar{w})^{\mathsf{L}}]w, \langle s[v], c\rangle)\}|_{\neg\mathsf{m}(v)} \\
\mathsf{env}(\bar{w}, w, \langle s, c\rangle) & := (\varepsilon, w^{!\mathsf{E}}, \langle \varepsilon, \{s\} \uplus c\rangle) \\
(\bar{w}_1, w_1, \langle \varepsilon, c_1\rangle) * (\bar{w}_2, w_2, \langle s_2, c_2\rangle) & := \{(\bar{w}_1\bar{w}_2, w, \langle s_2, c_1 \uplus c_2\rangle) \mid w \in w_1 \otimes w_2\}
\end{array}
$$

Here, the expression $w^{!\mathsf{E}}$ is defined as the relabeling of all steps in w to environment steps. It is straightforward to see that the operators are well-defined w.r.t. their specified signatures and that they are monotonic. Moreover, it is easy to see that $X * Y \in \mathsf{L}_S$ for $X, Y \in \mathsf{L}_S$. Therefore, we can use $*$ also as an operator of type $\mathsf{L}_S \times \mathsf{L}_S \to \mathsf{L}_S$ as done in the constraints for S.

Now we explain the constraints and operators: The main work is done by the $*$-operator which is generalized concatenation. It concatenates the same-level components of its operands, interleaves the macrostep components, and joins the reached configurations. A reaching triple in $R[p]$ is constructed by regarding a same-level path to some node $u \in \mathsf{N}_p$, represented by a summary triple from $S[u]$.[2] We distinguish the cases whether the local thread stays at node u ([REMPTY]) or whether it performs further macrosteps ([RCALL]). In the former case, we append the triple $(\varepsilon, \varepsilon, \{[u], \emptyset\})$. This sets the stack reached by the local thread to $[u]$. As we are not going to return from procedure p any more, we have to filter out triples that use monitors of procedure p in steps from spawned threads (environment steps). This is done by the $\cdot|_{\neg\mathsf{m}(p)}$-operation. Note that these monitors may be used in local steps, as prepending the call renders the subsequent uses to be reentering. In the latter case, we find a call edge of the form $(u, \mathsf{call}\ q, v)$, as macrosteps always start with a procedure call. The $(u, \mathsf{call}\ q, v); R[q]$-operation constructs a macrostep path from a reaching triple in $R[q]$, by adding the call edge and filtering out triples whose monitors conflict with the monitors $\mathsf{m}(v)$ held at the call site. Note that we also include a triple for the empty path in the result of the $(u, \mathsf{call}\ q, v);\cdot$-operator, in order to make it preserve the closure property. The [SEMPTY]-constraint accounts for the empty path from the beginning of a procedure and the [SBASE]- and [SCALL]-constraints propagate information over base and call edges respectively. The [SSPAWN]-constraint describes the effect of a spawn edge. The steps of the spawned thread are constructed from the $R[q]$-information. From the point of view of the thread executing the spawn edge, they are environment steps. The env-operation does the necessary L/E-relabeling. Note that the same-level components of triples in $R[q]$ are always empty, because due to our conventions a spawned procedure begins with a non-returning call. Hence, the env-operator ignores the same-level path component of its operand. By the well-known Knaster-Tarski fixpoint theorem the above constraint system has a least solution. In the following, R and S refer to the components of the least solution.

Theorem 8 (Correctness). *The least solution (R, S) is equal to the operational characterization, i.e. $R[p] = R^{\mathsf{op}}[p]$ and $S[u] = S^{\mathsf{op}}[u]$ for $p \in \mathsf{P}, u \in \mathsf{N}$.*

5 Abstractions

In this section, we develop an abstract interpretation of the constraint system over a finite domain that allows us to do conflict analysis by effective fixpoint computation. First, we briefly recall the concept of acquisition histories and describe our abstract domain and the abstract operators. We then analyze the running time of the resulting algorithm and show that the conflict detection problem is NP-complete.

Acquisition Histories. The concept of acquisition histories was introduced by Kahlon, Ivancic, and Gupta [8,6] to decide the interleavability of executions allocating locks in a well-nested fashion, but can also be applied to our reentrant

[2] Re-using the procedure summary information to describe initial segments of paths is a common technique to save redundant constraints.

monitors. The idea of acquisition histories is that two executions w_1 and w_2 are interleavable if and only if there is no *conflicting pair* of monitors m_1, m_2, that is w_1 enters m_1 and then uses m_2 and, vice versa, w_2 enters m_2 and then uses m_1. We define the set of acquisition histories by $\mathcal{H} := \{h : \mathcal{M} \to 2^{\mathcal{M}} \mid \forall m.\ h(m) = \emptyset \vee m \in h(m)\}$. Intuitively, an acquisition history maps all monitors m that are entered during an execution to the set of all monitors that are used after or in the same step as entering m. Hence we can define interleavability of two acquisition histories as $h_1 * h_2 :\Leftrightarrow \nexists m_1, m_2.\ m_2 \in h_1(m_1) \wedge m_1 \in h_2(m_2)$. In order to construct the acquisition history of a path backwards, we define the operator $\cdot;\cdot\ :\ 2^{\mathcal{M}} \times 2^{\mathcal{M}} \times \mathcal{H} \to \mathcal{H}$, that prepends a macrostep to an acquisition history: $((M_e, M_p); h)(m) :=$ if $m \in M_e$ then $M_e \cup M_p$ else $h(m)$. Intuitively, M_e is the set of monitors entered by the prepended macrostep and M_p is the set of monitors used in the whole path, including the prepended step. We define the acquisition history of a macrostep path by: $\alpha^{\mathsf{ah}}(\varepsilon) := \lambda m.\emptyset$ and $\alpha^{\mathsf{ah}}([l]w) := (\mathsf{ent}(l), \mathsf{pass}(l) \cup \mathsf{m}(w)); \alpha^{\mathsf{ah}}(w)$. We define a pointwise subset ordering on acquisition histories by $h \preceq h' :\Leftrightarrow \forall m.\ h(m) \subseteq h'(m)$. Obviously, this is an ordering and we have $h \preceq h' \wedge h' * h_2 \Rightarrow h * h_2$, i.e. a smaller acquisition history is interleavable with everything a bigger one is. The following theorem states that acquisition histories can be used to decide whether two paths are interleavable. It can be proven along the lines of [8].

Theorem 9. *For macrostep paths $w_1, w_2 \in \mathsf{MStep}^*$ we have $w_1 \otimes w_2 \neq \emptyset$ if and only if $\alpha^{\mathsf{ah}}(w_1) * \alpha^{\mathsf{ah}}(w_2)$.*

Abstract Domain. Let $U, V \subseteq \mathsf{N}$ be the two sets defining the conflict of interest. For a reaching triple, we record up to four abstract values from the set $\mathsf{D}^\sharp := \mathsf{D}_0^\sharp \cup \mathsf{D}_1^\sharp \cup \mathsf{D}_2^\sharp$, where $\mathsf{D}_0^\sharp := 2^{\mathcal{M}}$, $\mathsf{D}_1^\sharp := \mathcal{C}_1 \times (2^{\mathcal{M}})^3 \times \mathcal{H}$ with $\mathcal{C}_1 := \{\{U\}, \{V\}\}$, and $\mathsf{D}_2^\sharp := (2^{\mathcal{M}})^3$. While entries from $\mathsf{D}_0^\sharp$ are recorded for every reaching triple, entries from $\mathsf{D}_1^\sharp$ are recorded only for triples reaching one of the given sets U or V as specified by the first component, and entries from $\mathsf{D}_2^\sharp$ are recorded only for triples reaching a conflict. More specifically, the recorded information is specified by the abstraction functions $(\alpha_i :\ \mathsf{D} \to 2^{\mathsf{D}_i^\sharp})_{i=0,1,2}$:

$$
\begin{aligned}
&\alpha_0(\bar{w}, w, \langle s, c\rangle) := \{\mathsf{m}(\bar{w})\}\\
&\alpha_1(\bar{w}, w, \langle s, c\rangle) := \{(C, \mathsf{m}(\bar{w}), \mathsf{m}^{\mathsf{L}}(w), \mathsf{m}^{\mathsf{E}}(w), \alpha^{\mathsf{ah}}(w)) \mid C \in \mathcal{C}_1, \mathsf{at}_C(\langle s, c\rangle)\}\\
&\alpha_2(\bar{w}, w, \langle s, c\rangle) := \{(\mathsf{m}(\bar{w}), \mathsf{m}^{\mathsf{L}}(w), \mathsf{m}^{\mathsf{E}}(w)) \mid \mathsf{at}_{\{U,V\}}(\langle s, c\rangle)\}
\end{aligned}
$$

For $X \subseteq \mathsf{D}$, we define $\alpha_i(X) := \bigcup\{\alpha_i(x) \mid x \in X\}$. In order to treat entries from $\mathsf{D}_0^\sharp$,$\mathsf{D}_1^\sharp$,$\mathsf{D}_2^\sharp$ uniformly, we sometimes identify entries $M \in \mathsf{D}_0^\sharp$ with the tuple $(\emptyset, M, \emptyset, \emptyset, -)$ and entries $(\bar{M}, M_{\mathsf{L}}, M_{\mathsf{E}}) \in \mathsf{D}_2^\sharp$ with $(\{U, V\}, \bar{M}, M_{\mathsf{L}}, M_{\mathsf{E}}, -)$ and define $\mathcal{C} := \mathcal{C}_1 \cup \{\emptyset, \{U, V\}\}$. The symbol $-$ is a substitute for an acquisition history and we define $(M_e, M_p); - := -$ and agree that $- \preceq -$. Note that we never compare acquisition histories with $-$.

On $\mathsf{D}^\sharp$ we define an ordering $\leq$ by $(C, \bar{M}, M_{\mathsf{L}}, M_{\mathsf{E}}, h) \leq (C', \bar{M}', M'_{\mathsf{L}}, M'_{\mathsf{E}}, h')$ if and only if $C = C'$, $\bar{M} \subseteq \bar{M}'$, $M_{\mathsf{L}} \subseteq M'_{\mathsf{L}}$, $M_{\mathsf{E}} \subseteq M'_{\mathsf{E}}$, and $h \preceq h'$. Intuitively, $d < d'$ means that d reaches the same set $C \in \mathcal{C}$ of interesting nodes as d', but with weaker monitor requirements. Thus d' can be substituted by d in any context, and it is sufficient to collect the minimal elements when abstracting a set of reaching triples. Therefore we work with antichains. Formally, for an ordered set $(X, \leq)$ we write $(\mathsf{ac}(X), \sqsubseteq)$ for the complete lattice of antichains of X, i.e. $\mathsf{ac}(X) := \{M \subseteq X \mid \forall m, m' \in M.\ \neg m < m'\}$ and $M \sqsubseteq M' :\Leftrightarrow \forall m \in M. \exists m' \in M'.\ m \leq m'$. For an arbitrary set $M \subseteq X$, we write M^{ac} for the antichain reduction of M, i.e. the set of minimal elements of M. Note that $\cdot^{\mathsf{ac}}$ distributes over union and for $\mathcal{X} \subseteq \mathsf{ac}(X)$, the supremum of $\mathcal{X}$ is $\bigsqcup \mathcal{X} = (\bigcup \mathcal{X})^{\mathsf{ac}}$. Now we define our abstract domain by $\mathsf{L}^\sharp := \mathsf{ac}(\mathsf{D}^\sharp)$ and our abstraction $\alpha : \mathsf{L}_R \to \mathsf{L}^\sharp$ by $\alpha(X) = (\alpha_0(X) \cup \alpha_1(X) \cup \alpha_2(X))^{\mathsf{ac}}$. The abstraction α distributes over union, i.e. $\alpha(\bigcup \mathcal{X}) = \bigsqcup \{\alpha(X) \mid X \in \mathcal{X}\}$ for all $\mathcal{X} \subseteq \mathsf{L}_R$. Hence it is the lower adjoint of a Galois connection [12].

Example 10. Consider the reaching triples Tr_1 and Tr_2 introduced in Example 7. For $U := \{5\}$ and $V := \{9\}$, we have $\alpha(\{\mathsf{Tr}_1\}) = \{\{m_1\}\} \cup \{\mathsf{Tr}_1^\sharp\}$ with $\alpha_1(\mathsf{Tr}_1) = \{(\{V\}, \{m_1\}, \emptyset, \emptyset, \lambda m.\emptyset)\} =: \{\mathsf{Tr}_1^\sharp\}$ and $\alpha(\{\mathsf{Tr}_2\}) = \{\emptyset\} \cup \{\mathsf{Tr}_2^\sharp\}$ with $\alpha_1(\mathsf{Tr}_2) = \{(\{U\}, \emptyset, \emptyset, \{m_1, m_2\}, (m_1 \mapsto \{m_1, m_2\}))\} =: \{\mathsf{Tr}_2^\sharp\}$.

The $X * Y$-operation combines two reaching triples $t_1 := (\bar{w}_1, w_1, \langle \varepsilon, c_1 \rangle) \in X$ and $t_2 := (\bar{w}_2, w_2, \langle s, c_2 \rangle) \in Y$. The reached configuration of the resulting triples is $\langle s, c_1 \uplus c_2 \rangle$. For $C \in \mathcal{C}$, there are four cases for $\mathsf{at}_C(\langle s, c_1 \uplus c_2 \rangle)$. Either $\mathsf{at}_C(c_1)$, or $\mathsf{at}_C(c_2)$, or $C = \{U, V\}$ and $\mathsf{at}_{\{U\}}(c_1) \wedge \mathsf{at}_{\{V\}}(\langle s, c_2 \rangle)$ or $\mathsf{at}_{\{V\}}(c_1) \wedge \mathsf{at}_{\{U\}}(\langle s, c_2 \rangle)$. In the first case the interesting nodes are all reached by t_1. We then consider the triple $\tilde{t} := (\bar{w}_2, \varepsilon, \langle \tilde{s}, \tilde{c} \rangle) \in Y$ that exists due to the closure property of L_R. We have $\alpha(t_1 * \tilde{t}) = \alpha(\bar{w}_1 \bar{w}_2, w_1, \langle \tilde{s}, c_1 \uplus \tilde{c} \rangle) \sqsubseteq \alpha(t_1 * t_2)$. Thus for the abstraction of the result, we have to only consider $\alpha(t_1 * \tilde{t})$, which has the same acquisition history as t_1. Analogously, in the second case we only need to consider interleavings of the form $\tilde{t} * t_2$ for some $\tilde{t} = (\bar{w}_1, \varepsilon, \langle \varepsilon, \tilde{c} \rangle) \in X$. In the last two cases, the abstractions of both t_1 and t_2 contain the acquisition histories of w_1 and w_2, respectively. These can be used to check whether an interleaving exists. The abstraction of the resulting triples is then in $\mathsf{D}_2^\sharp$ (i.e. reaching a conflict) and thus contains no acquisition history. The operator $*^\sharp : \mathsf{L}^\sharp \times \mathsf{L}^\sharp \to \mathsf{L}^\sharp$ captures the ideas described above and is defined as the natural extension to antichains of the following definition: $(C_1, \bar{M}_1, M_{\mathsf{L}1}, M_{\mathsf{E}1}, h_1) *^\sharp (C_2, \bar{M}_2, M_{\mathsf{L}2}, M_{\mathsf{E}2}, h_2) := \{(C_i, \bar{M}_1 \cup \bar{M}_2, M_{\mathsf{L}i}, M_{\mathsf{E}i}, h_i) \mid i = 1, 2\}^{\mathsf{ac}} \sqcup \{(\{U, V\}, \bar{M}_1 \cup \bar{M}_2, M_{\mathsf{L}1} \cup M_{\mathsf{L}2}, M_{\mathsf{E}1} \cup M_{\mathsf{E}2}, -) \mid C_i = \{U\} \wedge C_{3-i} = \{V\} \wedge h_1 * h_2 \wedge i = 1, 2\}^{\mathsf{ac}}$. The definitions of the other abstract operators are straightforward (extended to antichains where necessary):

$$\begin{aligned}
&X|^\sharp_{\neg M} && := X \cap \{(\cdot, \cdot, \cdot, M_{\mathsf{E}}, \cdot) \in \mathsf{D}^\sharp \mid M_{\mathsf{E}} \cap M = \emptyset\} \\
&\mathsf{env}^\sharp((C, \bar{M}, M_{\mathsf{L}}, M_{\mathsf{E}}, h)) && := \{(C, \emptyset, \emptyset, M_{\mathsf{L}} \cup M_{\mathsf{E}}, h)\} \\
&(u, \mathsf{call}\ q, v);^\sharp (C, \bar{M}, M_{\mathsf{L}}, M_{\mathsf{E}}, h) && := \alpha(\varepsilon, \varepsilon, \langle [u], \emptyset \rangle) \sqcup \\
&\quad \{(C, \emptyset, \mathsf{m}(q) \cup \bar{M} \cup M_{\mathsf{L}}, M_{\mathsf{E}}, (\mathsf{m}(q), \bar{M} \cup M_{\mathsf{L}} \cup M_{\mathsf{E}}); h) \mid C \neq \emptyset\}|^\sharp_{\neg \mathsf{m}(v)}
\end{aligned}$$

Example 11. We show how our analysis utilizes acquisition histories to prevent detection of a spurious conflict between nodes 5 and 9 in the flowgraph of Fig. 2.

So let us assume $U := \{5\}$ and $V := \{9\}$. The combination of the paths to U and V is done by the [RCALL]-constraint for the edge $(2, \mathsf{call}\ q, 3)$. We consider the reaching triples $\mathsf{Tr}_1^\sharp \in R^\sharp[q]$ and $\mathsf{Tr}_2^\sharp \in S^\sharp[2]$ from Example 10. We have $\mathsf{Tr}_2^\sharp = (\{U\}, \emptyset, \emptyset, \{m_1, m_2\}, (m_1 \mapsto \{m_1, m_2\})) \in S^\sharp[2]|^\sharp_{\neg \mathsf{m}(2)}$ as $\mathsf{m}(2) = \emptyset$ and from $\mathsf{Tr}_1^\sharp$ we get $(\{V\}, \emptyset, \{m_1, m_2\}, \emptyset, (m_2 \mapsto \{m_1, m_2\})) \in (2, \mathsf{call}\ q, 3);^\sharp R^\sharp[q]$. However, the acquisition histories $h_1 := (m_1 \mapsto \{m_1, m_2\})$ and $h_2 := (m_2 \mapsto \{m_1, m_2\})$ are not interleavable $(\neg h_1 * h_2)$ because of the conflicting pair of monitors m_1, m_2 (i.e. $m_2 \in h_1(m_1)$ and $m_1 \in h_2(m_2)$). Therefore, these two entries are not combined by the $*^\sharp$-operator.

Lemma 12. *The abstract operators mirror the corresponding concrete operators precisely, i.e. for $X_R \in \mathsf{L}_R$ and $X_S \in \mathsf{L}_S$ the following holds:*

$$\begin{array}{ll} \alpha(X_S|_{\neg M}) = \alpha(X_S)|^\sharp_{\neg M} & \alpha((u, \mathsf{call}\ q, v); X_R) = (u, \mathsf{call}\ q, v);^\sharp \alpha(X_R) \\ \alpha(\mathsf{env}(X_R)) = \mathsf{env}^\sharp(\alpha(X_R)) & \alpha(X_S * X_R) = \alpha(X_S) *^\sharp \alpha(X_R) \end{array}$$

The proof for the $*$-operator follows the ideas described above and is omitted here due to the limited space. The proofs for the other operators are straightforward.

Theorem 13. *Let $(R^\sharp, S^\sharp)$ be the least solution of the constraint system interpreted over the abstract domain $\mathsf{L}^\sharp$ using the abstract operators and replacing the constants by their abstractions. It exactly matches the least solution (R, S) of the concrete constraint system, i.e. $\forall p \in \mathsf{P}.\ \alpha(R[p]) = R^\sharp[p]$ and $\forall u \in \mathsf{N}.\ \alpha(S[u]) = S^\sharp[u]$. It can be computed in time $O((|\mathsf{N}| + |\mathsf{E}|) \cdot 2^{\mathsf{poly}(|\mathcal{M}|)})$.*

Theorem 13 follows from Lemma 12 by standard results of abstract interpretation, see, e.g. [3,5].

Corollary 14. *Conflict analysis can be done in time $O((|\mathsf{N}| + |\mathsf{E}|) \cdot 2^{\mathsf{poly}(|\mathcal{M}|)})$, i.e. linear in the program size and exponential in the number of monitors.*

Proof. From Theorems 3, 8, 13, the definitions of R^{op} and α, and the convention that the main-procedure starts with a non-returning call, it is straightforward to show that $\exists c.\ \{[\mathsf{e}_{\mathsf{main}}]\} \xrightarrow{*} c \wedge \mathsf{at}_{\{U,V\}}(c)$ if and only if $R^\sharp[\mathsf{e}_{\mathsf{main}}] \cap \mathsf{D}_2^\sharp \neq \emptyset$. Thus, an algorithm for conflict analysis can compute the least solution of the abstract constraint system and check whether $R^\sharp[\mathsf{e}_{\mathsf{main}}]$ contains an entry from $\mathsf{D}_2^\sharp$. $\square$

Theorem 15. *Deciding whether a given flowgraph has a conflict is NP-complete.*

Proof. We sketch the NP-hardness direction here, that justifies the exponential running time of our algorithm. We show NP-hardness of the reachability problem for a single control node (that can obviously be reduced to conflict detection) by a reduction from 3SAT. For a formula in conjunctive normal form (CNF) $\bigwedge_{1 \le i \le n} \bigvee_{1 \le j \le 3} l_{ij}$ with $l_{ij} \in \{x_1, \bar{x}_1, ..., x_m, \bar{x}_m\}$, we construct the following program[3] with the procedures main, $p_1, ..., p_{m+1}$, c and monitors $x_1, \bar{x}_1, ..., x_m, \bar{x}_m$:

[3] The translation of this textual representation to our model is straightforward.

```
proc main {call $p_1$}
proc $p_i$ /* $1 \leq i \leq m$ */ {
  {sync ($x_i$) {call $p_{i+1}$}} OR {sync ($\bar{x}_i$) {call $p_{i+1}$}} }
proc $p_{m+1}$ {spawn c; loop forever}
proc c {
  {sync ($l_{11}$){skip}} OR {sync ($l_{12}$){skip}} OR {sync ($l_{13}$){skip}};
  ...
  {sync ($l_{n1}$){skip}} OR {sync ($l_{n2}$){skip}} OR {sync ($l_{n3}$){skip}};
  u: // Control node that is checked to be reachable }
```

The statement `sync` (m) `{ ... }` denotes a block synchronized on monitor m and `OR` denotes nondeterministic choice. Intuitively, the procedures p_1 to p_m guess the values of the variables, where `sync` (x_i) corresponds to setting x_i to false, and vice versa, `sync` $(\bar{x}_i)$ corresponds to setting x_i to true. Finally, procedure q checks whether the clauses are satisfied. Control node u is reachable if and only if the formula is satisfiable.

This construction exploits dynamic thread creation and uses a procedure that cannot terminate. One can do similar constructions for the simultaneous reachability of two control nodes in a setting with two fixed threads where all procedures eventually terminate. Thus conflict analysis is also NP-hard for models like the one used in [8]. However, for the model of [8] reachability of a single program point is decidable in polynomial time which highlights the inherent complexity of thread creation.

6 Conclusion

In this paper we studied conflict analysis for a program model with procedure calls, dynamic thread creation and synchronization via reentrant monitors. We showed that conflict analysis is NP-complete. We then used the concept of restricted schedules to come to grips with the arbitrary interleaving between threads. We showed that every reachable configuration is also reachable by an execution with a restricted schedule. We developed a constraint system based characterization of restricted executions, and used abstract interpretation to derive an algorithm for conflict checking that is linear in the program size and exponential only in the number of monitors.

We have developed a formal proof of a similar approach to conflict analysis [10] in Isabelle/HOL [14]. The formalization of the flowgraphs, operational semantics, restricted schedules and acquisition histories are the same as in this paper. The constraint systems follow similar ideas but the abstract constraint systems there are justified directly w.r.t. to the operational semantics instead of using abstract interpretation. Moreover, the height of the abstract domain quadratically depends on the number of procedures. The NP-completeness result has not been formalized in Isabelle/HOL.

Further research required on this topic includes the following: Our algorithm is exponential in the number of monitors. However, for real programs, the nesting depth of monitors is usually significantly smaller than their number. There is strong evidence that this observation can be exploited to design a more efficient analysis. A similar effect was also described in [8] for a model with a fixed set of threads. Furthermore, our algorithm is only able to check for conflicts — while this is an important practical problem, there are other interesting problems like bitvector analysis or high-level data races [1], which may be tackled by generalizing our approach. In order to apply our algorithm to languages with dynamic referencing of monitors (like Java), a preceeding pointer analysis is required. The combination of our analysis with such analyses has to be investigated.

Acknowledgment. We thank Ahmed Bouajjani, Javier Esparza, Nicholas Kidd, Thomas Reps, Helmut Seidl, Bernhard Steffen, Dejvuth Suwimonteerabuth, and Tayssir Touili for interesting discussions, and the anonymous referees for very helpful comments.

References

1. Artho, C., Havelund, K., Biere, A.: High-level data races. In: Isaías, P.T., Sedes, F., Augusto, J.C., Ultes-Nitsche, U. (eds.) NDDL/VVEIS, 2003-08-21, pp. 82–93. ICEIS Press (2003) ISBN:972-09916-2-6
2. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, Springer, Heidelberg (2005)
3. Cousot, P.: Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. Electronic Notes in Theoretical Computer Science 6 (1997), `www.elsevier.nl/locate/entcs/volume6.html`
4. Esparza, J., Podelski, A.: Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In: Proc. of POPL 2000, pp. 1–11. Springer, Heidelberg (2000)
5. Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. Journal of the ACM 47(2), 361–416 (2000)
6. Kahlon, V., Gupta, A.: An automata-theoretic approach for model checking threads for LTL properties. In: Proc. of LICS 2006, pp. 101–110. IEEE Computer Society, Los Alamitos (2006)
7. Kahlon, V., Gupta, A.: On the analysis of interacting pushdown systems. In: POPL, pp. 303–314 (2007)
8. Kahlon, V., Ivancic, F., Gupta, A.: Reasoning about threads communicating via locks. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 505–518. Springer, Heidelberg (2005)
9. Lammich, P., Müller-Olm, M.: Conflict analysis of programs with procedures, dynamic thread creation, and monitors. Technical Report, `http://cs.uni-muenster.de/u/mmo/pubs/`
10. Lammich, P., Müller-Olm, M.: Formalization of conflict analysis of programs with procedures, thread creation, and monitors. In: Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs (December 2007), `http://afp.sourceforge.net/entries/Program-Conflict-Analysis.shtml` (Formal proof development)
11. Lammich, P., Müller-Olm, M.: Precise fixpoint-based analysis of programs with thread-creation. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR. LNCS, vol. 4703, pp. 287–302. Springer, Heidelberg (2007)

12. Melton, A., Schmidt, D.A., Strecker, G.E.: Galois connections and computer science applications. In: Poigné, A., Pitt, D.H., Rydeheard, D.E., Abramsky, S. (eds.) Category Theory and Computer Programming. LNCS, vol. 240, pp. 299–312. Springer, Heidelberg (1986)
13. Müller-Olm, M.: Precise interprocedural dependence analysis of parallel programs. Theor. Comput. Sci. 311(1-3), 325–388 (2004)
14. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
15. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. TOPLAS 22(2), 416–430 (2000)
16. Seidl, H., Steffen, B.: Constraint-Based Inter-Procedural Analysis of Parallel Programs. Nordic Journal of Computing (NJC) 7(4), 375–400 (2000)
17. Yahav, E.: Verifying safety properties of concurrent Java programs using 3-valued logic. ACM SIGPLAN Notices 36(3), 27–40 (2001)

Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis

Elvira Albert[1], Puri Arenas[1], Samir Genaim[2], and Germán Puebla[2]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

Abstract. The classical approach to automatic cost analysis consists of two phases. Given a program and some measure of cost, we first produce *recurrence relations* (RRs) which capture the cost of our program in terms of the size of its input data. Second, we convert such RRs into *closed form* (i.e., without recurrences). Whereas the first phase has received considerable attention, with a number of cost analyses available for a variety of programming languages, the second phase has received comparatively little attention. In this paper we first study the features of RRs generated by automatic cost analysis and discuss why existing computer algebra systems are not appropriate for automatically obtaining closed form solutions nor upper bounds of them. Then we present, to our knowledge, the first practical framework for the fully automatic generation of reasonably accurate upper bounds of RRs originating from cost analysis of a wide range of programs. It is based on the inference of *ranking functions* and *loop invariants* and on *partial evaluation*.

1 Introduction

The aim of *cost analysis* is to obtain *static* information about the execution cost of programs w.r.t. some cost *measure*. Cost analysis has a large application field, which includes resource certification [11,4,16,9], whereby code consumers can reject code which is not guaranteed to run within the resources available. The resources considered include processor cycles, memory usage, or *billable events*, e.g., the number of text messages or bytes sent on a mobile network.

A well-known approach to automatic cost analysis, which dates back to the seminal work of [25], consists of two phases. In the first phase, given a program and some cost measure, we produce a set of equations which captures the cost of our program in terms of the size of its input data. Such equations are generated by converting the iteration constructs of the program (loops and recursion) into recurrences and by inferring *size relations* which approximate how the size of arguments varies. This set of equations can be regarded as *recurrence relations* (RRs for short). Equivalently, it can be regarded as *time bound programs* [22]. The aim of the second phase is to obtain a non-recursive representation of the equations, known as *closed form*. In most cases, it is not possible to find an exact solution and the closed form corresponds to an upper bound.

There are a number of cost analyses available which are based on this approach and which can handle a range of programming languages, including functional [7,18,22,23,24,25], logic [12,20], and imperative [1,3]. While in all such analyses the first phase is studied in detail, the second phase has received comparatively less attention. Basically, there are three different approaches for the second phase. One approach, which is conceptually linked viewing equations as time bound programs, was proposed in [18] and advocated in [22]. It is based on existing source-to-source transformations which convert recursive programs into non-recursive ones. The second approach consists in building restricted recurrence solvers using standard mathematical techniques, as in [12,25]. The third approach consists in relying on existing *computer algebra systems* (CASs for short) such as Mathematica®, MAXIMA, MAPLE, etc., as in [3,7,23,24].

The problem with the three approaches above is that they assume a rather limited form of equations which does not cover the essential features of equations actually generated by automatic cost analysis. In the rest of the paper, we will concentrate on viewing equations as recurrence relations and will use the term *Cost Relation* (CR for short) to refer to the relations produced by automatic cost analysis. In our own experience with [3], we have detected that existing CASs are, in most cases, not capable of handling CRs. We argue that automatically converting CRs into the format accepted by CASs is unfeasible. Furthermore, even in those cases where CASs can be used, the solutions obtained are so complicated that they become useless for most practical purposes. An altogether different approach to cost analysis is based on type systems with resource annotations which does not use equations. Thus, it does not need to obtain closed forms, but it is typically restricted to linear bounds [16]. The need for improved mechanisms for obtaining upper bounds was already pointed out in Hickey and Cohen [14]. A relevant work in this direction is PURRS [5], which has been the first system to provide, in a fully automatic way, non-asymptotic upper and lower bounds for a wide class of recurrences. Unfortunately, and unlike our proposal, it also requires CRs to be deterministic. Marion et. al. [19,8] use a kind of polynomial ranking functions, but the approach is limited to polynomial bounds and can only handle a rather restricted form of CRs.

We believe that the lack of automatic tools for the above second phase is a major reason for the diminished use of automatic cost analysis. In this paper we study the features of CRs and discuss why existing CASs are not appropriate for automatically bounding them. Furthermore, we present, to our knowledge, the first practical framework for the fully automatic inference of reasonably accurate upper bounds for CRs originating from a wide range of programs. To do this, we apply semantic-based transformation and analysis techniques, including inference of *ranking functions*, *loop invariants* and the use of *partial evaluation*.

1.1 Motivating Example

Example 1. Consider the Java code in Fig. 1. It uses a class `List` for (non sorted) linked lists of integers. Method `del` receives an input list without repetitions `l`,

```
void del(List l, int p, int a[], int la, int b[], int lb){
   while (l!=null) {
      if (l.data<p) {
         la=rm_vec(l.data, a, la);
      } else {
          lb=rm_vec(l.data, b, lb);
      }
      l=l.next;
   }
}

int rm_vec(int e, int a[], int la){
   int i=0;
   while (i<la && a[i]<e)  i++;
   for (int j=i; j<la−1; j++) a[j]=a[j+1];
   return la−1;
}
```

(1) $Del(l,a,la,b,lb)=1+C(l,a,la,b,lb)$
$\quad\{b\geq lb, lb\geq 0, a\geq la, la\geq 0, l\geq 0\}$
(2) $C(l,a,la,b,lb)=2\ \{a\geq la, b\geq lb, b\geq 0, a\geq 0, l=0\}$
(3) $C(l,a,la,b,lb)=$
$\quad 25+D(a,la,0)+E(la,j)+C(l',a,la-1,b,lb)$
$\quad\{a\geq 0, a\geq la, b\geq lb, j\geq 0, b\geq 0, l>l', l>0\}$
(4) $C(l,a,la,b,lb)=$
$\quad 24+D(b,lb,0)+E(lb,j)+C(l',a,la,b,lb-1)$
$\quad\{b\geq 0, b\geq lb, a\geq la, j\geq 0, a\geq 0, l>l', l>0\}$
(5) $D(a,la,i)=3\quad\{i\geq la, a\geq la, i\geq 0\}$
(6) $D(a,la,i)=8\quad\{i<la, a\geq la, i\geq 0\}$
(7) $D(a,la,i)=10+D(a,la,i+1)\ \{i<la, a\geq la, i\geq 0\}$
(8) $E(la,j)=5\quad\{j\geq la-1, j\geq 0\}$
(9) $E(la,j)=15+E(la,j+1)\quad\{j<la-1, j\geq 0\}$

Fig. 1. Java Code and the Result of Cost Analysis

an integer value p (the *pivot*), two sorted arrays of integers a and b, and two integers la and lb which indicate, respectively, the number of positions occupied in a and b. The array a (resp. b) is expected to contain values which are smaller than the pivot p (resp. greater or equal). Under the assumption that all values in l are contained in either a or b, the method del removes all values in l from the corresponding arrays. The auxiliary method rm_vec removes a given value e from an array a of length la and returns its new length, la−1.

We have applied the cost analysis in [3] on this program in order to approximate the cost of executing the method del in terms of the number of executed bytecode instructions. For this, we first compile the program to bytecode and then analyze the resulting bytecode. Fig. 1 (right) presents the results of analysis, after performing *partial evaluation*, as we will explain in Sec. 6, and inlining equality constraints (e.g., inlining equality $lb'=lb-1$ is done by replacing the occurrences of lb' by $lb-1$). In the analysis results, the data structures in the program are abstracted to their sizes: l represents the maximal *path-length* [15] of the corresponding dynamic structure, which in this case corresponds to the length of the list, a and b are the lengths of the corresponding arrays, and la and lb are the integer values of the corresponding variables. There are nine equations which define the relation Del, which corresponds to the cost of the method del, and three auxiliary recursive relations, C, D, and E. Each of them corresponds to a loop (C: while loop in del; D: while loop in rm_vec; and E: for loop in rm_vec). Each equation is annotated with a set of constraints which capture *size relations* between the values of variables in the left hand side (lhs) and those in the right hand side (rhs). In addition, size relations may contain applicability conditions (i.e., *guards*) by providing constraints which only affect variables in the lhs. Let us explain the equations for D. Eqs. (5) and (6) are base cases which correspond to the exits from the loop when i≥la and a[i]≥e, respectively. Note that the condition a[i]≥e does not appear in the size relation of Eq. (6) nor (7). This is because the array a has been abstracted to its length. Thus, the value in a[i] is no longer observable. For our cost measure , we count 3 bytecode instructions in Eq. (5) and 8 in Eq. (6). The cost of executing an iteration of the loop is captured by Eq. (7), where the condition $i{<}la$ must be satisfied and variable i is increased by one at each recursive call. □

1.2 Cost Relations vs. Recurrence Relations

CRs differ from standard RRs in the following ways:

(a) *Non-determinism.* In contrast to RRs, CRs are possibly non-deterministic: equations for the same relation are not required to be mutually exclusive. Even if the programming language is deterministic, *size abstractions* introduce a loss of precision: some guards which make the original program deterministic may not be observable when using the size of arguments instead of their actual values. In Ex. 1, this happens between Eqs. (3) and (4) and also between (6) and (7).

(b) *Inexact size relations.* CRs may have size relations which contain constraints (not equalities). When dealing with realistic programming languages which contain non-linear data structures, such as trees, it is often the case that size analysis does not produce exact results. E.g., analysis may infer that the size of a data structure strictly decreases from one iteration to another, but it may be unable to provide the precise reduction. This happens in Ex. 1 in Eqs. (3) and (4).

(c) *Multiple arguments.* CRs usually depend on several arguments that may increase (variable i in Eq. (7)) or decrease (variable l in Eq. (2)) at each iteration. In fact, the number of times that a relation is executed can be a combination of several of its arguments. E.g., relation E is executed $la{-}j{-}1$ times.

Point (a) was detected already in [25], where an explicit `when` operator is added to the RR language to introduce non-determinism, but no complete method for handling it is provided. Point (b) is another source of non-determinism. As a result, CRs do not define functions, but rather relations. Given a relation C and input values $\overline{v}$, there may exist multiple results for $C(\overline{v})$. Sometimes it is possible to automatically convert relations with several arguments into relations with only one. However, in contrast to our approach, it is restricted to very simple cases such as when the CR only count constant cost expressions.

Existing methods for solving RRs are insufficient to bound CRs since they do not cover points (a), (b), and (c) above. On the other hand, CASs can solve complex recurrences (e.g., coefficients to function calls can be polynomials) which our framework cannot handle. However, this additional power is not needed in cost analysis, since such recurrences do not occur as the result of cost analysis.

An obvious way of obtaining upper bounds in non-deterministic CRs would be to introduce a maximization operator. Unfortunately, such operator is not supported by existing CAS. Adding it is far from trivial, since computing the maximum when the equations are not mutually exclusive requires taking into account multiple possibilities, which results in a highly combinatorial problem. Another possibility is to convert CRs into RRs. For this, we need to remove equations from CRs as well as sometimes to replace inexact size relations by exact ones while preserving the worst-case solution. However, this is not possible in general. E.g., in Fig. 1, the maximum cost is obtained when the execution interleaves Eqs. (3) and (4), and therefore we cannot remove either of them.

2 Cost Relations: Evaluation and Upper Bounds

Let us introduce some notation. We use x, y, z, possibly subscripted, to denote variables which range over integers ($\mathbb{Z}$), v, w denote integer values, a, b natural numbers ($\mathbb{N}$) and q rational numbers ($\mathbb{Q}$). We denote by $\mathbb{Q}^+$ (resp. $\mathbb{R}^+$) the set of non-negative rational (resp. real) numbers. We use $\bar{t}$ to denote a sequence of entities $t_1, \ldots, t_n$, for some $n{>}0$. We sometimes apply set operations on sequences. Given $\overline{x}$, an *assignment* for $\overline{x}$ is a sequence $\overline{v}$ (denoted by $[\overline{x}/\overline{v}]$). Given any entity t, $t[\overline{x}/\overline{v}]$ stands for the result of replacing in t each occurrence of x_i by v_i. We use $vars(t)$ to refer to the set of variables occurring in t. A *linear expression* has the form $q_0{+}q_1x_1{+}\cdots{+}q_nx_n$. A *linear constraint* has the form l_1 *op* l_2 where l_1 and l_2 are linear expressions and $op \in \{=, \leq, <, >, \geq\}$. A *size relation* φ is a set of linear constraints (interpreted as a conjunction). The operator $\bar{\exists}\overline{x}.\varphi$ eliminates from φ all variables except for $\overline{x}$. We write $\varphi_1 \models \varphi_2$ to indicate that φ_1 implies φ_2. The following definition presents our notion of *basic cost expression.*

Definition 1 (basic cost expression). Basic cost expressions *are of the form:* $\mathsf{exp}{::=}a|\mathsf{nat}(l)|\mathsf{exp}{+}\mathsf{exp}|\mathsf{exp}{*}\mathsf{exp}|\mathsf{exp}^a|\log_{\mathsf{a}}(\mathsf{exp})|a^{\mathsf{exp}}|\max(S)|\frac{\mathsf{exp}}{a}|\mathsf{exp}{-}a$*, where* $a{\geq}1$*,* l *is a linear expression,* S *is a non empty set of cost expressions,* $\mathsf{nat}{:}\mathbb{Z}{\rightarrow}\mathbb{Q}^+$ *is defined as* $\mathsf{nat}(v){=}\max(\{v,0\})$*, and* exp *satisfies that for any assignment* $\overline{v}$ *for* $vars(\mathsf{exp})$ *we have that* $\mathsf{exp}[vars(\mathsf{exp})/\overline{v}] \in \mathbb{R}^+$*.*

Basic cost expressions are symbolic expressions which indicate the resources we accumulate and are the non-recursive building blocks for defining cost relations. They enjoy two crucial properties: (1) by definition, they are always evaluated to non negative values; (2) replacing a sub-expression $\mathsf{nat}(l)$ by $\mathsf{nat}(l')$ such that $l'{\geq}l$, results in an upper bound of the original expression.

A *cost relation* C of arity n is a subset of $\mathbb{Z}^n * \mathbb{R}^+$. This means that for a single tuple $\overline{v}$ of integers there can be multiple solutions in $C(\overline{v})$. We use C and D to refer to cost relations. Cost analysis of a program usually produces multiple, interconnected, cost relations. We refer to such sets of cost relations as *cost relation systems* (CRSs for short), which we formally define below.

Definition 2 (Cost Relation System). *A* cost relation system $\mathcal{S}$ *is a set of equations of the form* $\langle C(\overline{x}){=}\mathsf{exp}{+}\sum_{i=0}^{k} D_i(\overline{y}_i),\ \varphi\rangle$ *with* $k{\geq}0$*, where* C *and all* D_i *are cost relations, all variables* $\overline{x}$ *and* $\overline{y}_i$ *are distinct variables;* exp *is a basic cost expression; and* φ *is a size relation between* $\bar{x}$ *and* $\bar{x}{\cup}vars(\mathsf{exp}){\cup}\bar{y}_i$*.*

In contrast to standard definitions of RRs, the variables which occur in the rhs of the equations in CRSs do not need to be related to those in the lhs by equality constraints. Other constraints such as $\leq$ and $<$ can also be used. We denote by $rel(\mathcal{S})$ the set of cost relations which are defined in $\mathcal{S}$. Also, $def(\mathcal{S}, C)$ denotes the subset of the equations in $\mathcal{S}$ whose lhs is of the form $C(\overline{x})$. W.l.o.g. we assume that all equations in $def(\mathcal{S}, C)$ have the same variable names in the lhs. We assume that any CRS $\mathcal{S}$ is self-contained in the sense that all cost relations which appear in the rhs of an equation in $\mathcal{S}$ must be in $rel(\mathcal{S})$.

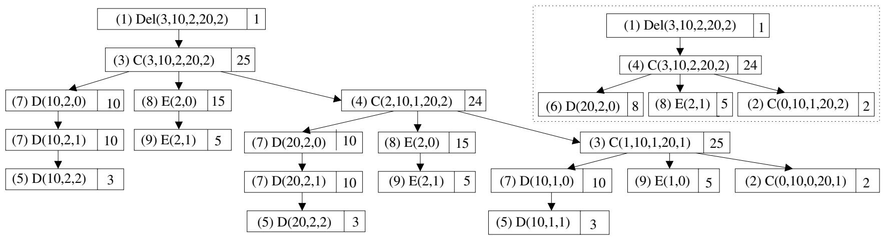


Fig. 2. Two Evaluation Trees for $Del(3, 10, 2, 20, 2)$

We now provide a semantics for CRSs. Given a CRS $\mathcal{S}$, a *call* is of the form $C(\overline{v})$, where $C{\in}rel(\mathcal{S})$ and $\overline{v}$ are integer values. Calls are evaluated in two phases. In the first phase, we build an *evaluation tree* for the call. In the second phase we obtain a *value* in $\mathbb{R}^+$ by adding up the constants which appear in the nodes of the evaluation tree. We make evaluation trees explicit since, as discussed below, our approximation techniques are based on reasoning about the number of nodes and the values in the nodes in such evaluation trees. Evaluation trees are obtained by repeatedly expanding nodes which contain calls to relations. Each expansion is performed w.r.t an appropriate instantiation of a rhs of an applicable equation. If all leaves in the tree contain basic cost expressions then there is no node left to expand and the process terminates. We will represent evaluation trees using nested terms of the form $node(Call, Local\_Cost, Children)$, where $Local\_Cost$ is a constant in $\mathbb{R}^+$ and $Children$ is a sequence of evaluation trees.

Definition 3 (evaluation tree). *Given a CRS* $\mathcal{S}$ *and a call* $C(\overline{v})$*, a tree node* $(C(\overline{v}), e, \langle T_1, \ldots, T_k\rangle)$ *is an* evaluation tree *for* $C(\overline{v})$ *in* $\mathcal{S}$*, denoted* $\mathsf{Tree}(C(\overline{v}), \mathcal{S})$ *if: 1) there is a renamed apart equation* $\langle C(\overline{x}){=}\mathtt{exp}{+}\sum_{i=0}^{k} D_i(\overline{y}_i),\ \varphi\rangle \in \mathcal{S}$ *s.t.* φ' *is satisfiable in* $\mathbb{Z}$*, with* $\varphi'{=}\varphi[\overline{x}/\overline{v}]$*, and 2) there exist assignments* $\overline{w}, \overline{v}_i$ *for* $vars(\mathtt{exp}), \overline{y}_i$ *respectively s.t.* $\varphi'[vars(\mathtt{exp})/\overline{w},\ \overline{y}_i/\overline{v}_i]$ *is satisfiable in* $\mathbb{Z}$*, and 3)* $e{=}\mathtt{exp}[vars(\mathtt{exp})/\overline{w}]$*,* T_i *is an evaluation tree* $\mathsf{Tree}(D_i(\overline{v}_i), \mathcal{S})$ *with* $i = 0, \ldots, k$*.*

In step 1 we look for an equation $\mathcal{E}$ which is applicable for solving $C(\overline{v})$. Note that there may be several equations which are applicable. In step 2 we look for assignments for the variables in the rhs of $\mathcal{E}$ which satisfy the size relations associated to $\mathcal{E}$. This a non-deterministic step as there may be (infinitely many) different assignments which satisfy all size relations. Finally, in step 3 we apply the assignment to `exp` and continue recursively evaluating the calls. We use $Trees(C(\overline{v}), \mathcal{S})$ to denote the set of all evaluation trees for $C(\overline{v})$. We define $Answers(C(\overline{v}), \mathcal{S}){=}\{\mathsf{Sum}(T) \mid T{\in}Trees(C(\overline{v}), \mathcal{S})\}$, where $\mathsf{Sum}(T)$ traverses all nodes in T and computes the sum of the cost expressions in them.

Example 2. Fig. 2 shows two possible evaluation trees for $Del(3, 10, 2, 20, 2)$. The tree on the left has maximal cost, whereas the one on the right has minimal cost. A node in either tree contains a call (left box) and its local cost (right box) and it is linked by arrows to its children. We annotate calls with a number in

(2) $C(l, a, la, b, lb) = \boxed{2}$
$\{a \geq la, b \geq lb, b \geq 0, a \geq 0, l = 0\}$
(3) $C(l, a, la, b, lb) =$
$\boxed{38+15*\mathsf{nat}(la\text{-}j\text{-}1)+10*\mathsf{nat}(la)} + C(l', a, la-1, b, lb)$
$\{a \geq 0, a \geq la, b \geq lb, j \geq 0, b \geq 0, l > l', l > 0\}$
(4) $C(l, a, la, b, lb) =$
$\boxed{37+15*\mathsf{nat}(lb\text{-}j\text{-}1)+10*\mathsf{nat}(lb)} + C(l', a, la, b, lb-1)$
$\{b \geq 0, b \geq lb, a \geq la, j \geq 0, a \geq 0, l > l', l > 0\}$

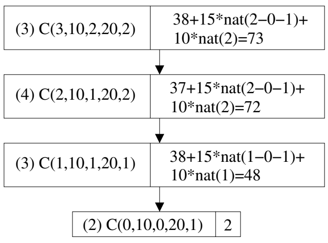

(3) C(3,10,2,20,2)
38+15*nat(2−0−1)+ 10*nat(2)=73
(4) C(2,10,1,20,2)
37+15*nat(2−0−1)+ 10*nat(2)=72
(3) C(1,10,1,20,1)
38+15*nat(1−0−1)+ 10*nat(1)=48
(2) C(0,10,0,20,1)
2

Fig. 3. Self-Contained CR for relation C and a corresponding evaluation tree

parenthesis to indicate the equation which was selected for evaluating such call. Note that, in the recursive call to C in Eqs. (3) and (4), we are allowed to pick any value l' s.t. $l' < l$. In the tree on the left we always assign $l' = l-1$. This is what happens in actual executions of the program. In the tree on the right we assign $l' = l-3$ in the recursive call to C. The latter results in a minimal approximation, however, it does not correspond to any actual execution. This is a side effect of using safe approximations in static analysis: information is correct in the sense that at least one of the evaluation trees must correspond to the actual cost, but there may be other trees with different cost. In fact, there are an infinite number of evaluation trees for our example call, as step 2 can provide an infinite number of assignments to variable j which are compatible with the constraint $j \geq 0$ in Eqs. (3) and (4). This shows that approaches like [13] based on evaluation of CRSs are not of general applicability. Nevertheless, it is possible to find an upper bound for this call since though the number of trees is infinite, infinitely many of them produce equivalent results. □

2.1 Closed Form Upper Bounds for Cost Relations

Let C be a relation over $\mathbb{Z}^n * \mathbb{R}^+$. A function $U : \mathbb{Z}^n \to \mathbb{R}^+$ is an *upper bound* of C iff $\forall \overline{v} \in \mathbb{Z}^n$, $\forall a \in Answers(C(\overline{v}), \mathcal{S})$, $U(\overline{v}) \geq a$. We use C^+ to refer to an upper bound of C. A function $f : \mathbb{Z}^n \to \mathbb{R}^+$ is in *closed form* if it is defined as $f(\overline{x}) = \mathtt{exp}$, with $\mathtt{exp}$ a basic cost expression s.t. $vars(\mathtt{exp}) \subseteq \overline{x}$. An important feature of CRSs, inherited from RRs, is their *compositionality*, which allows computing upper bounds of CRSs by concentrating on one relation at a time. I.e., given a cost equation for $C(\overline{x})$ which calls $D(\overline{y})$, we can replace the call to $D(\overline{y})$ by $D^+(\overline{y})$. The resulting relation is trivially an upper bound of the original one. E.g., suppose that we have the following upper bounds: $E^+(la, j) = 5+15*\mathsf{nat}(la-j-1)$ and $D^+(a, la, i) = 8+10*\mathsf{nat}(la-i)$. Replacing the calls to D and E in equations (3) and (4) by D^+ and E^+ results in the CRS shown in Fig. 3.

The compositionality principle only results in an effective mechanism if all recursions are *direct* (i.e., all cycles are of length one). In that case we can start by computing upper bounds for cost relations which do not depend on any other relations, which we refer to as *standalone cost relations* and continue by replacing the computed upper bounds on the equations which call such relations. In the following, we formalize our method by assuming standalone cost relations and in Sec. 6 we provide a mechanism for obtaining direct recursion automatically.

Existing approaches to compute upper bounds and asymptotic complexity of RRs, usually applied by hand, are based on reasoning about evaluation trees in terms of their size, depth, number of nodes, etc. They typically consider two categories of nodes: (1) *internal* nodes, which correspond to applying recursive equations, and (2) *leaves* of the tree(s), which correspond to the application of a base (non-recursive) case. The central idea then is to count (or obtain an upper bound on) the number of leaves and the number of internal nodes in the tree separately and then multiply each of these by an upper bound on the cost of the base case and of a recursive step, respectively. For instance, in the evaluation tree in Fig. 3 for the standalone cost relation C, there are three internal nodes and one leaf. The values in the internal nodes, once performed the evaluation of the expressions are 73, 72, and 48, therefore 73 is the worst case. In the case of leaves, the only value is 2. Therefore, the tightest upper bound we can find using this approximation is $3{\times}73{+}1{*}2{=}221 \geq 73{+}72{+}48{+}2{=}193$.

We now extend the approximation scheme mentioned above in order to consider all possible evaluation trees which may exist for a call. In the following, we use $|S|$ to denote the cardinality of a set S. Also, given an evaluation tree T, $\mathit{leaf}(T)$ denotes the set of leaves of T (i.e., those without children) and $\mathit{internal}(T)$ denotes the set of internal nodes (all nodes but the leaves) of T.

Proposition 1 (node-count upper bound). *Let C be a cost relation and let $C^+(\overline{x}) = \mathit{internal}^+(\overline{x}) * \mathit{costr}^+(\overline{x}) + \mathit{leaf}^+(\overline{x}) * \mathit{costnr}^+(\overline{x})$, where $\mathit{internal}^+(\overline{x})$, $\mathit{costr}^+(\overline{x})$, $\mathit{leaf}^+(\overline{x})$ and $\mathit{costnr}^+(\overline{x})$ are closed form functions defined on $\mathbb{Z}^n \to \mathbb{R}^+$. Then, C^+ is an upper bound of C if for all $\overline{v} \in \mathbb{Z}^n$ and for all $T \in \mathit{Trees}(C(\overline{v}), \mathcal{S})$, it holds: (1) $\mathit{internal}^+(\overline{v}) \geq |\mathit{internal}(T)|$ and $\mathit{leaf}^+(\overline{v}) \geq |\mathit{leaf}(T)|$; (2) $\mathit{costr}^+(\overline{v})$ is an upper bound of $\{e \mid \mathit{node}(\_, e, \_) \in \mathit{internal}(T)\}$ and (3) $\mathit{costnr}^+(\overline{v})$ is an upper bound of $\{e \mid \mathit{node}(\_, e, \_) \in \mathit{leaf}(T)\}$.*

3 Upper Bounds on the Number of Nodes

In this section we present an automatic mechanism to obtain safe $\mathit{internal}^+(\overline{x})$ and $\mathit{leaf}^+(\overline{x})$ functions which are valid for any assignment for $\overline{x}$. The basic idea is to first obtain upper bounds b and $h^+(\overline{x})$ on, respectively, the *branching factor* and *height* (the distance from the root to the deepest leaf) of all corresponding evaluation trees, and then use the number of internal nodes and leaves of a *complete* tree with such branching factor and height as an upper bound. Then,

$$\mathit{leaf}^+(\overline{x}) = b^{h^+(\overline{x})} \qquad \mathit{internal}^+(\overline{x}) = \begin{cases} h^+(\overline{x}) & b{=}1 \\ \frac{b^{h^+(\overline{x})}-1}{b-1} & b{\geq}2 \end{cases}$$

For a cost relation C, the branching factor b in any evaluation tree for a call $C(\overline{v})$ is limited by the maximum number of recursive calls which occur in a single equation for C. We now propose a way to compute an upper bound for the height, h^+. Given an evaluation tree $T \in \mathit{Trees}(C(\overline{v}), \mathcal{S})$ for a cost relation C, consecutive nodes in any branch of T represent consecutive recursive calls which occur during the evaluation of $C(\overline{v})$. Therefore, bounding the height of a tree may be reduced to bounding consecutive recursive calls. The notion of *loop* in a cost relation, which we introduce below, is used to model consecutive calls.

Definition 4. *Let* $\mathcal{E} = \langle C(\overline{x}) = \mathtt{exp} + \sum_{i=1}^{k} C(\overline{y_i}), \varphi \rangle$ *be an equation for a cost relation* C*. Then,* $\mathit{Loops}(\mathcal{E}) = \{\langle C(\overline{x}) \rightarrow C(\bar{y}_i), \varphi' \rangle \mid \varphi' = \overline{\exists} \bar{x} \cup \bar{y}_i . \varphi, i = 1 \cdots k\}$ *is the set of* loops *induced by* $\mathcal{E}$*. Similarly,* $\mathit{Loops}(C) = \cup_{\mathcal{E} \in \mathit{def}(\mathcal{S}, C)} \mathit{Loops}(\mathcal{E})$*.*

Example 3. Eqs. (3) and (4) in Fig. 3 induce the following two loops:

$$(3)\langle C(l,a,la,b,lb) \rightarrow C(l',a,la',b,lb), \varphi_1' = \{a \geq 0, a \geq la, b \geq lb, b \geq 0, l > l', l > 0, la' = la - 1\}\rangle$$
$$(4)\langle C(l,a,la,b,lb) \rightarrow C(l',a,la,b,lb'), \varphi_2' = \{b \geq 0, b \geq lb, a \geq la, a \geq 0, l > l', l > 0, lb' = lb - 1\}\rangle$$

Bounding the number of consecutive recursive calls is extensively used in the context of termination analysis. It is usually done by proving that there is a function f from the loop's arguments to a *well-founded* partial order which decreases in any two consecutive calls and which guarantees the absence of infinite traces, and thus termination. These functions are usually called *ranking functions*. We propose to use the ranking function to generate a h^+ function. In practice, we use [21] to generate functions which are defined as follows: a function $f{:}\mathbb{Z}^n \mapsto \mathbb{Z}$ is a *ranking function* for a loop $\langle C(\bar{x}) \rightarrow C(\bar{y}), \varphi \rangle$ if $\varphi \models f(\bar{x}) > f(\bar{y})$ and $\varphi \models f(\bar{x}) \geq 0$.

Example 4. The function $f_C(l, a, la, b, lb) = l$ is a ranking function for C in the cost relation in Fig. 3. Note that φ_1' and φ_2' in the above loops of C contain the constraints $\{l > l', l > 0\}$ which is enough to guarantee that f_C is decreasing and well-founded. The height of the evaluation tree for $C(3, 10, 2, 20, 2)$ is precisely predicted by $f_C(3, 10, 2, 20, 2) = 3$. Ranking functions may involve several arguments, e.g., $f_D(a, la, i) = la - i$ is a ranking function for $\langle D(a, la, i) \rightarrow D(a, la, i'), \{i' = i + 1, i < la, a \geq la, i \geq 0\}\rangle$ which comes from Eq. (7). □

Observe that the use of global ranking functions allows bounding the number of iterations of possibly non-deterministic CRSs with multiple arguments (see Sec. 1.2). In order to be able to define h^+ in terms of the ranking function, one thing to fix is that the ranking function might return a negative value when is applied to values which correspond to base cases (leaves of the tree). Therefore, we define $h^+(\overline{x}) = \mathtt{nat}(f_C(\overline{x}))$. Function $\mathtt{nat}$ guarantees that negative values are lifted to 0 and, therefore, they provide a correct approximation for the height of evaluation trees with a single node. Even though the ranking function provides an upper bound for the height of the corresponding trees, in some cases we can further refine it and obtain a tighter upper bound. For example, if the difference between the value of the ranking function in each two consecutive calls is larger than a constant $\delta > 1$, then $\lceil \mathtt{nat}(\frac{f_C(\bar{x})}{\delta}) \rceil$ is a tighter upper bound. A more interesting case, if each loop $\langle C(\overline{x}) \rightarrow C(\overline{y}), \varphi \rangle \in \mathit{Loops}(C)$ satisfies $\varphi \models f_C(\overline{x}) \geq k * f_C(\overline{y})$ where $k > 1$, then the height of the tree is bounded by $\lceil \log_k(\mathtt{nat}(f_C(\overline{v}) + 1)) \rceil$.

4 Estimating the Cost Per Node

Consider the evaluation tree in Fig. 3. Note that all expressions in the nodes are instances of the expressions which appear in the corresponding equations. Thus, computing $costr^+(\overline{x})$ and $costnr^+(\overline{x})$ can be done by first finding an upper bound of such expressions and then combining them through a *max* operator. We first compute *invariants* for the values that the expression variables can take w.r.t. the initial values, and use them to derive upper bounds for such expressions.

4.1 Invariants

Computing an *invariant* (in terms of linear constraints) that holds in all *calling contexts* (*contexts* for short) to a relation C between the arguments at the initial call and at each call during the evaluation can be done by using $Loops(C)$. Intuitively, if we know that a linear constraint ψ holds between the arguments of the initial call $C(\overline{x}_0)$ and those of a recursive call $C(\overline{x})$, denoted $\langle C(\overline{x}_0) \rightsquigarrow C(\overline{x}), \psi\rangle$, and we have a loop $\langle C(\overline{x}) \rightarrow C(\overline{y}), \varphi\rangle \in Loops(C)$, then we can apply the loop one more step and get the new *calling context* $\langle C(\overline{x}_0) \rightsquigarrow C(\overline{y}), \bar{\exists}\overline{x_0} \cup \overline{y}.\psi \wedge \varphi\rangle$.

Definition 5 (loop invariants). *For a relation C, let $\mathcal{T}$ be an operator defined:*

$$\mathcal{T}(X) = \left\{ \langle C(\overline{x}_0) \rightsquigarrow C(\overline{y}), \psi'\rangle \;\middle|\; \begin{array}{l} \langle C(\overline{x}_0) \rightsquigarrow C(\overline{x}), \psi\rangle \in X, \langle C(\overline{x}) \rightarrow C(\overline{y}), \varphi\rangle \in Loops(C), \\ \psi' = \bar{\exists}\overline{x_0} \cup \overline{y}.\psi \wedge \varphi \end{array} \right\}$$

which derives a set of contexts, from a given context X, by applying all loops, then the loop invariants I is $lfp \cup_{i \geq 0} \mathcal{T}^i(I_0)$ where $I_0 = \{\langle C(\overline{x}_0) \rightsquigarrow C(\overline{x}), \{\overline{x}_0 = \overline{x}\}\rangle\}$.

Example 5. Let us compute I for the loops in Sec. 3. The initial context is $I_1 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l = l_0, a = a_0, la = la_0, b = b_0, lb = lb_0\}\rangle$ where $\bar{x}_0 = \langle l_0, a_0, la_0, b_0, lb_0\rangle$ and $\bar{x} = \langle l, a, la, b, lb\rangle$. In the first iteration we compute $\mathcal{T}^0(\{I_1\})$ which by definition is $\{I_1\}$. In the second iteration we compute $\mathcal{T}^1(\{I_1\})$ which results in

$$\begin{array}{l} I_2 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{\underline{l < l_0}, a = a_0, \underline{la = la_0 - 1}, b = b_0, lb = lb_0, \underline{l_0 > 0}\}\rangle \\ I_3 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{\underline{l < l_0}, a = a_0, la = la_0, b = b_0, \underline{lb = lb_0 - 1}, \underline{l_0 > 0}\}\rangle \end{array}$$

where I_2 and I_3 correspond to applying respectively the first loop and second loops on I_1. The underlined constraints are the modifications due to the application of the loop. Note that in I_2 the variable la_0 decreases by one, and in I_3 lb_0 decreases by one. The third iteration $\mathcal{T}^2(\{I_1\})$, i.e. $\mathcal{T}(\{I_2, I_3\})$, results in

$$\begin{array}{l} I_4 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l < l_0, a = a_0, \underline{la = la_0 - 2}, b = b_0, lb = lb_0, l_0 > 0\}\rangle \\ I_5 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l < l_0, a = a_0, la = la_0 - 1, b = b_0, \underline{lb = lb_0 - 1}, l_0 > 0\}\rangle \\ I_6 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l < l_0, a = a_0, la = la_0, b = b_0, \underline{lb = lb_0 - 2}, l_0 > 0\}\rangle \\ I_7 = \langle C(\bar{x}_0) \rightsquigarrow C(\bar{x}), \{l < l_0, a = a_0, \underline{la = la_0 - 1}, b = b_0, lb = lb_0 - 1, l_0 > 0\}\rangle \end{array}$$

where I_4 and I_5 originate from applying the loops to I_2, and I_6 and I_7 from applying the loops to I_3. The modifications on the constraints reflect that, when applying a loop, either we decrease la or lb. After three iterations, the invariant I includes $I_1 \cdots I_7$. More iterations will add more contexts that further modify the value of la or lb. Therefore, the invariant I grows indefinitely in this case. □

In practice, we approximate I using abstract interpretation over, for instance, the domain of convex polyhedra [10], whereby we obtain the invariant $\Psi{=}\langle C(\overline{x}_0) \leadsto C(\overline{x}), \{l{\leq}l_0, a{=}a_0, la{\leq}la_0, b{=}b_0, lb{\leq}lb_0\}\rangle$.

4.2 Upper Bounds on Cost Expressions

Once invariants are available, finding upper bounds of cost expressions can be done by maximizing their nat parts independently. This is possible due to the monotonicity property of cost expressions. Consider, for example, the expression $\mathsf{nat}(la{-}j{-}1)$ which appears in equation (3) of Fig. 3. We want to infer an upper bound of the values that it can be evaluated to in terms of the input values $\langle l_0, a_0, la_0, b_0, lb_0\rangle$. We have inferred, in Sec. 4.1, that whenever we call C the invariant Ψ holds, from which we can see that the maximum value that la can take is la_0. In addition, from the local size relations φ of equation (3) we know that $j{\geq}0$. Since $la{-}j{-}1$ takes its maximal value when la is maximal and j is minimal, the expression $la_0{-}1$ is an upper bound for $la{-}j{-}1$. This can be done automatically using linear constraints tools [6]. Given a cost equation $\langle C(\overline{x}){=}\mathtt{exp}{+}\sum_{i=0}^{k} C(\overline{y}_i),\ \varphi\rangle$ and an invariant $\langle C(\overline{x}_0){\leadsto}C(\overline{x}), \Psi\rangle$, the function below computes an upper bound for exp by maximizing its nat components.

```
1: function $ub\_exp(\mathtt{exp}, \overline{x}_0, \varphi, \Psi)$
2:     mexp=exp
3:     for all $\mathsf{nat}(f){\in}\mathtt{exp}$ do
4:         $\Psi'{=}\bar{\exists}\overline{x}_0, r.(\varphi{\wedge}\Psi{\wedge}(r{=}f))$        // $r$ is a fresh variable
5:         if $\exists f'$ s.t. $vars(f'){\subseteq}\overline{x}_0$ and $\Psi'{\models}r{\leq}f'$ then $\mathtt{mexp}{=}\mathtt{mexp}[\mathsf{nat}(f)/\mathsf{nat}(f')]$
6:         else return $\infty$
7:     return mexp
```

This function computes an upper bound f' for each expression f which occurs inside a nat operator and then replaces in exp all such f expressions with their corresponding upper bounds (line 5). If it cannot find an upper bound, the method returns ∞ (line 6). The *ub_exp* function is complete in the sense that if Ψ and φ imply that there is an upper bound for a given $\mathsf{nat}(f)$, then we can find one by *syntactically* looking on Ψ' (line 4).

Example 6. Applying *ub_exp* to $\mathtt{exp}_3$ and $\mathtt{exp}_4$ of Eqs. (3) and (4) in Fig. 3 w.r.t. the invariant we have computed in Sec. 4.1 results in $\mathtt{mexp}_3{=}38{+}15{*}\mathsf{nat}(la_0{-}1)$ $+10{*}\mathsf{nat}(la_0)$ and $\mathtt{mexp}_4{=}37{+}15{*}\mathsf{nat}(lb_0{-}1) + 10{*}\mathsf{nat}(lb_0)$. □

Theorem 1. *Let $\mathcal{S}{=}\mathcal{S}_1{\cup}\mathcal{S}_2$ be a cost relation where $\mathcal{S}_1$ and $\mathcal{S}_2$ are respectively the sets of non-recursive and recursive equations for C, and let $\mathcal{I}{=}\langle C(\overline{x}_0) {\leadsto}C(\overline{x}), \Psi\rangle$ be a loop invariant for C; $E_i{=}\{ub\_exp(\mathtt{exp}, \overline{x}_0, \varphi, \Psi) \mid \langle C(\overline{x}) = \mathtt{exp}{+} \sum_{j=0}^{k} C(\overline{y}_j), \varphi\rangle{\in}\mathcal{S}_i\}$; $costnr^{+}(\overline{x}_0){=}\max(E_1)$ and $costr^{+}(\overline{x}_0){=}\max(E_2)$. Then for any call $C(\overline{v})$ and for all $T \in Trees(C(\overline{v}), \mathcal{S})$: (1) $\forall node(\_, e, \_){\in}internal(T)$ we have $costr^{+}(\overline{v}){\geq}e$; and (2) $\forall node(\_, e, \_){\in}leaf(T)$ we have $costnr^{+}(\overline{v}){\geq}e$.*

Example 7. At this point we have all the pieces in order to compute an upper bound for the CRS depicted in Fig. 1 as described in Prop. 1. We start by computing upper bounds for E and D as they are cost relations:

	Ranking Function	*costnr*$^+$	*costr*$^+$	*Upper Bound*
$E(la_0, j_0)$	$\mathsf{nat}(la_0 - j_0 - 1)$	5	15	$5+15*\mathsf{nat}(la_0 - j_0 - 1)$
$D(a_0, la_0, i_0)$	$\mathsf{nat}(la_0 - i_0)$	8	10	$8+10*\mathsf{nat}(la_0 - i_0)$

These upper bounds can then be substituted in the equations (3) and (4) which results in the cost relation for C depicted in Fig. 3. We have already computed a ranking function for C in Ex. 4 and *costnr*$^+$ and *costr*$^+$ in Ex. 6, which are then combined into $C^+(l_0, a_0, la_0, b_0, lb_0)=2+\mathsf{nat}(l_0)*max(\{\mathtt{mexp}_3, \mathtt{mexp}_4\})$. Reasoning similarly, for *Del* we get the upper bound shown in Table 1. □

5 Improving Accuracy in Divide and Conquer Programs

For some CRSs, we can obtain a more accurate upper bound by approximating the cost of *levels* instead of approximating the cost of nodes, as indicated by Prop. 1. Given an evaluation tree T, we denote by $\mathsf{Sum\_Level}(T, i)$ the sum of the values of all nodes in T which are at depth i, i.e., at distance i from the root.

Proposition 2 (level-count upper bound). *Let C be a cost relation and let C^+ be a function defined as: $C^+(\overline{x})=l^+(\overline{x}) * costl^+(\overline{x})$, where $l^+(\overline{x})$ and $costl^+(\overline{x})$ are closed form functions defined on $\mathbb{Z}^n \rightarrow \mathbb{R}^+$. Then, C^+ is an upper bound of C if for all $\overline{v} \in \mathbb{Z}^n$ and $T \in Trees(C(\overline{v}), \mathcal{S})$, it holds: (1) $l^+(\overline{v}) \geq depth(T) + 1$; and (2) $\forall i \in \{0, \ldots, depth(T)\}$ we have that $costl^+(\overline{v}) \geq \mathsf{Sum\_Level}(T, i)$.*

The function l^+ can simply be defined as $l^+(\overline{x})=\mathsf{nat}(f_C(\overline{x}))+1$ (see Sec. 3). Finding an accurate *costl*$^+$ function is not easy in general, which makes Prop. 2 not as widely applicable as Prop. 1. However, evaluation trees for *divide and conquer* programs satisfy that $\mathsf{Sum\_Level}(T, k) \geq \mathsf{Sum\_Level}(T, k+1)$, i.e., the cost per level does not increase from one level to another. In that case, we can take the cost of the root node as an upper bound of $costl^+(\overline{x})$. A sufficient condition for a cost relation falling into the divide and conquer class is that each cost expression that is contributed by an equation is greater than or equal to the sum of the cost expressions contributed by the corresponding immediate recursive calls. This check is implemented in our prototype using [6].

Consider a CRS with the two equations $\langle C(n)=0, \{n \leq 0\}\rangle$ and $\langle C(n) = \mathsf{nat}(n)+C(n_1)+C(n_2), \varphi\rangle$ where $\varphi=\{n>0, n_1+n_2+1 \leq n, n \geq 2*n_1, n \geq 2*n_2, n_1 \geq 0, n_2 \geq 0\}$. It corresponds to a divide and conquer problem such as merge-sort. In order to prove that $\mathsf{Sum\_Level}$ does not increase, it is enough to check that, in the second equation, n is greater than or equal to the sum of the expressions that immediately result from the calls $C(n_1)$ and $C(n_2)$, which are n_1 and n_2 respectively. This can be done by simply checking that $\varphi \models n \geq n_1+n_2$. Then, $costl^+(\overline{x})=max\{0, \mathsf{nat}(x)\}=\mathsf{nat}(x)$. Thus, given that $l^+(x)=\lceil \log_2(\mathsf{nat}(x)+1)\rceil+1$, we obtain the upper bound $\mathsf{nat}(x)*(\lceil \log_2(\mathsf{nat}(x)+1)\rceil+1)$. Note that by using the node-count approach we would obtain $\mathsf{nat}(x)*(2^{\mathsf{nat}(x)}-1)$ as upper bound.

6 Direct Recursion Using Partial Evaluation

Automatically generated CRSs often contain recursions which are not direct, i.e., cycles involve more than one function. E.g., the actual CRS obtained for the program in Fig. 1 by the analysis in [3] differs from that shown in the right hand side of Fig. 1 in that, instead of Eqs. (8) and (9), the "for" loop results in:

(8')	$E(la,j)=5+F(la,j,j',la')$	$\{j'=j, la'=la-1, j'\geq 0\}$
(9')	$F(la,j,j',la')=H(j',la')$	$\{j'\geq la'\}$
(10)	$F(la,j,j',la')=G(la,j,j',la')$	$\{j'<la'\}$
(11)	$H(j',la')=0$	$\{\}$
(12)	$G(la,j,j',la')=10+E(la,j+1)$	$\{j<la-1, j\geq 0, la-la'=1, j'=j\}$

Now, E captures the cost of the loop condition "$j<la-1$" (5 cost units) plus the cost of its continuation, captured by F. Eq. (9') corresponds to the exit of the loop (it calls H, Eq. (11), which has 0 cost). Eq. (10) captures the cost of one iteration by calling G, Eq. (12), which accumulates 10 units and returns to E.

In this section we present an automatic transformation of CRSs into *directly recursive* form. The transformation is based on *partial evaluation* (PE) [17] and it is performed by replacing calls to intermediate relations by their definitions using *unfolding*. The first step in the transformation is to find a *binding time classification* (or BTC for short) which declares which relations are *residual*, i.e., they have to remain in the CRS. The remaining relations are considered *unfoldable*, i.e., they are eliminated. For computing BTCs, we associate to each CRS $\mathcal{S}$ a *call graph*, denoted $\mathcal{G}(\mathcal{S})$, which is the directed graph obtained from $\mathcal{S}$ by taking $rel(\mathcal{S})$ as the set of nodes and by including an arc (C, D) iff D appears in the rhs of an equation for C. The following definition provides sufficient conditions on a BTC which guarantee that we obtain a directly recursive CRS.

Definition 6. *Let $\mathcal{G}(\mathcal{S})$ be the call graph of $\mathcal{S}$ and let SCC be its strongly connected components. A BTC* btc *for $\mathcal{S}$ is* directly recursive *if for all $S\in SCC$ the following conditions hold: (1) if $s_1, s_2\in S$ and $s_1, s_2\in\mathsf{btc}$, then $s_1=s_2$; and (2) if S has a cycle, then there exists $s\in S$ such that $s\in\mathsf{btc}$.*

Condition 1 ensures that all recursions in the transformed CRS are direct, as there is only one residual relation per SCC. Condition 2 guarantees that the unfolding process terminates, as there is a residual relation per cycle. A directly recursive BTC for the above example is $\mathsf{btc}=\{E\}$. In our implementation we only include in the BTC the *covering point* (i.e., a node which is part of all cycles) of SCCs which contain cycles, but no node is included for SCCs without cycles. This way of computing BTCs, in addition to ensuring direct recursion, also eliminates all relations which are not part of cycles (such as H in our example).

We now define unfolding in the context of CRSs. Such unfolding is guided by a BTC and at each step it combines both cost expressions and size relations.

Definition 7 (unfolding). *Given a CRS $\mathcal{S}$, a call $C(\overline{x}_0)$ s.t. $C\in rel(\mathcal{S})$, a size relation $\varphi_{\overline{x}_0}$ over $\overline{x}_0$, and a BTC* btc *for $\mathcal{S}$, a pair $\langle E,\varphi\rangle$ is an* unfolding *for $C(\overline{x}_0)$ and $\varphi_{\overline{x}_0}$ in $\mathcal{S}$ w.r.t.* btc*, denoted $\mathsf{Unfold}(\langle C(\overline{x}_0),\varphi_{\overline{x}_0}\rangle,\mathcal{S},\mathsf{btc})\rightsquigarrow\langle E,\varphi\rangle$, if either of the following conditions hold:*

(res) $C\in\mathsf{btc}\wedge\varphi\neq\mathsf{true}\wedge\langle E,\varphi\rangle=\langle C(\overline{x}_0),\varphi_{\overline{x}_0}\rangle$;

(unf) $(C\notin\mathsf{btc}\vee\varphi=\mathsf{true})\wedge\langle E,\varphi\rangle=\langle(\mathsf{exp}+e_1+\ldots+e_k),\varphi' \bigwedge_{i=1..k}\varphi_i\rangle$

where $\langle C(\overline{x})=\mathsf{exp}+\sum_{i=1}^{k}D_i(\overline{y}_i),\varphi_C\rangle$ is a renamed apart equation in $\mathcal{S}$ s.t. $\varphi'=\varphi_{\overline{x}_0}\wedge\varphi_C[\overline{x}/\overline{x}_0]$ is satisfiable in $\mathbb{Z}$ and $\forall 1\leq i\leq k$ $\mathsf{Unfold}(\langle D_i(\overline{y_i}),\varphi'\rangle,\mathcal{S},\mathsf{btc})\rightsquigarrow\langle e_i,\varphi_i\rangle$.

The first case, (**res**), is required for termination. When we call a relation C which is marked as residual, we simply return the initial call $C(\overline{x}_0)$ and size relation $\varphi_{\overline{x}_0}$, as long as the current size relation $\varphi_{\overline{x}_0}$ is not the initial one (true). The latter condition is added in order to force the initial unfolding step for relations marked as residual. In all subsequent calls to Unfold different from the initial one, the size relation is different from true. The second case (**unf**) corresponds to continuing the unfolding process. Note that step 1 is non-deterministic, since often cost relations contain several equations. Since expressions are transitively unfolded, step 2 may also provide multiple solutions. Also, if the final size relation φ is unsatisfiable, we simply do not regard $\langle E, \varphi\rangle$ as a valid unfolding.

Example 8. Given the initial call $\langle E(la, j), true\rangle$, we obtain an unfolding by performing the following steps, denoted by $\overset{e}{\leadsto}$ where e is the selected equation:
$\langle E(la,j), true\rangle \overset{(8')}{\leadsto} \langle 5{+}F(la,j,j',la'), \{j'{=}j, la'{=}la{-}1, j'{\geq}0\}\rangle \overset{(10)}{\leadsto}$
$\langle 5{+}G(la,j,j',la'), \{j'{=}j, la'{=}la{-}1, j'{\geq}0, j'{<}la'\}\rangle \overset{(12)}{\leadsto} \langle 15{+}E(la,j''), \{j{<}la{-}1, j{\geq}0\}\rangle$
The call $E(la, j'')$ is not further unfolded as E belongs to btc and $\varphi{\neq}true$. □

From each result of unfolding we can build a *residual equation*. Given the unfolding $\mathsf{Unfold}(\langle C(\overline{x}_0), \varphi_{\overline{x}_0}\rangle, \mathcal{S}, \mathsf{btc}) {\leadsto} \langle E, \varphi\rangle$ its corresponding residual equation is $\langle C(\overline{x}_0){=}E, \varphi\rangle$. As customary in PE, a *partial evaluation* of C is obtained by collecting all residual equations for the call $\langle C(\overline{x}_0), \mathsf{true}\rangle$. The PE of $\langle E(la, j), true\rangle$ results in Eqs. (8) and (9) of Fig. 1. Eq. (9) is obtained from the unfolding steps depicted in Ex. 8 and Eq. (8) from unfolding w.r.t. Eqs. (8'), (9'), and (11).

Correctness of PE ensures that the solutions of CRSs are preserved. Regarding completeness, we can obtain direct recursion if all SCCs in the call graph have covering point(s). Importantly, structured loops (for, while, etc.) and recursive patterns found in most programs result in CRSs that satisfy this property. In addition, before applying PE, we check that the CRS terminates [2] with respect to the initial query, otherwise we might compromise non-termination and thus lead to incorrect upper bounds. We believe this check is not required when CRSs are generated from imperative programs.

7 Experiments in Cost Analysis of Java Bytecode

A prototype implementation in Ciao Prolog, which uses PPL [6] for manipulating linear constraints, is available at `http://www.cliplab.org/Systems/PUBS`. We have performed a series of experiments which are shown in Table 1. We have used CRSs automatically generated by the cost analyzer of Java bytecode described in [3] using two cost measures: heap consumption for those marked with "*", and the number of executed bytecode instructions for the rest. The benchmarks are presented in increasing complexity order and grouped by asymptotic class. Those marked with M were solved using Mathematica® by [3] but after significant human intervention. The marks a, b and c after the name indicate, respectively, if the CRS is non-deterministic, has inexact size relations and multiple arguments (Sec. 1.2). Column $\#_{eq}$ shows the number of equations before PE (in brackets

Table 1. Experiments on Cost Analysis of Java Bytecode

Benchmark		$\#_{eq}$	**T**	$\#^c_{eq}$	$\mathbf{T}_{pe}$	$\mathbf{T}_{ub}$	**Rat.**	**Upper Bound**
Polynomial*	abc	23 (3)	13	346 (70)	174	649	2.4	216
DivByTwo	ab	9 (3)	3	323 (68)	166	596	2.4	$8\log_2(\mathsf{nat}(2x-1)+1)+14$
Factorial$^{\mathsf{M}}$		8 (2)	4	314 (66)	165	590	2.4	$9\mathsf{nat}(x)+4$
ArrayRev$^{\mathsf{M}}$	a	9 (3)	4	305 (64)	165	579	2.4	$14\mathsf{nat}(x)+12$
Concat$^{\mathsf{M}}$	ac	14 (5)	13	296 (62)	158	538	2.4	$11\mathsf{nat}(x)+11\mathsf{nat}(y)+25$
Incr$^{\mathsf{M}}$	ac	28 (5)	29	282 (58)	155	490	2.3	$19\mathsf{nat}(x+1)+9$
ListRev$^{\mathsf{M}}$	abc	9 (3)	4	254 (54)	144	415	2.2	$13\mathsf{nat}(x)+8$
MergeList	abc	21 (4)	18	245 (52)	138	406	2.2	$29\mathsf{nat}(x+y)+26$
Power		8 (2)	3	223 (48)	125	371	2.2	$10\mathsf{nat}(x)+4$
Cons*	ab	22 (2)	6	214 (46)	123	359	2.3	$22\mathsf{nat}(x-1)+24$
EvenDigits	abc	18 (5)	9	191 (44)	115	322	2.3	$\mathsf{nat}(x)(8\log_2(\mathsf{nat}(2x-3)+1)+24)+9\mathsf{nat}(x)+9$
ListInter	abc	37 (9)	59	173 (40)	110	298	2.4	$\mathsf{nat}(x)(10\mathsf{nat}(y)+43)+21$
SelectOrd	ac	19 (6)	27	136 (32)	86	198	2.1	$\mathsf{nat}(x-2)(17\mathsf{nat}(x-2)+34)+9$
FactSum	a	17 (5)	8	117 (27)	76	173	2.1	$\mathsf{nat}(x+1)(9\mathsf{nat}(x)+16)+6$
Delete	abc	33 (9)	125	100 (23)	71	165	2.4	$3+\mathsf{nat}(l)\max(38+15\mathsf{nat}(la-1)+10\mathsf{nat}(la),$ $37+15\mathsf{nat}(lb-1)+10\mathsf{nat}(lb))$
MatMult$^{\mathsf{M}}$	ac	19 (7)	23	67 (15)	27	40	1.0	$\mathsf{nat}(y)(\mathsf{nat}(x)(27\mathsf{nat}(x))+10)+17$
Hanoi		9 (2)	4	48 (8)	23	17	0.8	$20(2^{\mathsf{nat}(x)})$-17
Fibonacci$^{\mathsf{M}}$		8 (2)	5	39 (6)	20	13	0.8	$18(2^{\mathsf{nat}(x-1)})$-13
BST*	ab	31 (4)	26	31 (4)	19	7	0.9	$96(2^{\mathsf{nat}(x)})$-49

after PE). Note that PE greatly reduces $\#_{eq}$ in all benchmarks. Column **T** shows the total runtime in milliseconds. The experiments have been performed on an Intel Core 2 Duo 1.86GHz with 2GB of RAM, running Linux.

The next four columns aim at demonstrating the scalability of our approach. To do so, we connect the CRSs for the different benchmarks by introducing a call from each CRS to the one appearing immediately below it in the table. Such call is always introduced in a recursive equation. Column $\#^c_{eq}$ shows the number of equations we want to solve in each case (in brackets after PE). Reading this column bottom-up, we can see that BST has the same number of equations as the original one and that, progressively, each benchmark adds its own number of equations to $\#^c_{eq}$. Thus, in the first row we have a CRS with all the equations connected, i.e., we compute an upper bound of CRS with at least 19 nested loops and 346 equations. The total runtime is split into $\mathbf{T}_{pe}$ and $\mathbf{T}_{ub}$, where $\mathbf{T}_{pe}$ is the time of PE and it shows that even though PE is a *global* transformation, its time efficiency is linear with the number of equations. Our system solves 346 equations in $823ms$. Column **Rat**. shows the total time per equation. The ratio is small for benchmarks with few equations, and for reasonably large CRSs (from Delete upwards) it almost has no variation (2.1–2.4 ms/eq). The small increase is due to the fact that the equations count more complex expressions as we connect more benchmarks. This demonstrates that our approach is totally scalable, even if the implementation is preliminary. The upper bound expressions get considerably large when the benchmarks are composed together. We are currently implementing standard techniques for simplification of arithmetic expressions.

Acknowledgments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT* project, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project. S. Genaim was supported by a *Juan de la Cierva* Fellowship awarded by MEC.

References

1. Adachi, A., Kasai, T., Moriya, E.: A theoretical study of the time analysis of programs. In: Becvar, J. (ed.) MFCS 1979. LNCS, vol. 74, pp. 201–207. Springer, Heidelberg (1979)
2. Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., Zanardini, D.: Termination Analysis of Java Bytecode. In: Proc. of FMOODS. LNCS, Springer, Heidelberg (to appear, 2008)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of java bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, Springer, Heidelberg (2007)
4. Aspinall, D., Gilmore, S., Hofmann, M., Sannella, D., Stark, I.: Mobile Resource Guarantees for Smart Devices. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, Springer, Heidelberg (2005)
5. Bagnara, R., Pescetti, A., Zaccagnini, A., Zaffanella, E.: PURRS: Towards computer algebra support for fully automatic worst-case complexity analysis. Technical report, arXiv:cs/0512056 (2005), http://arxiv.org/
6. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.M.: Possibly not closed convex polyhedra and the Parma Polyhedra Library. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, Springer, Heidelberg (2002)
7. Benzinger, R.: Automated higher-order complexity analysis. In: TCS, vol. 318(1-2) (2004)
8. Bonfante, G., Marion, J.-Y., Moyen, J.-Y.: Quasi-interpretations and small space bounds. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, Springer, Heidelberg (2005)
9. Chander, A., Espinosa, D., Islam, N., Lee, P., Necula, G.: Enforcing resource bounds via static verification of dynamic checks. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, Springer, Heidelberg (2005)
10. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL (1978)
11. Crary, K., Weirich, S.: Resource bound certification. In: POPL (2000)
12. Debray, S.K., Lin, N.W.: Cost analysis of logic programs. TOPLAS 15(5) (1993)
13. Gómez, G., Liu, Y.A.: Automatic time-bound analysis for a higher-order language. In: PEPM, ACM Press, New York (2002)
14. Hickey, T., Cohen, J.: Automating program analysis. J. ACM 35(1) (1988)
15. Hill, P.M., Payet, E., Spoto, F.: Path-length analysis of object-oriented programs. In: Proc. EAAI, Elsevier, Amsterdam (2006)
16. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: POPL (2003)
17. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice Hall, New York (1993)
18. Le Metayer, D.: ACE: An Automatic Complexity Evaluator. TOPLAS 10(2) (1988)

19. Marion, J.-Y., Péchoux, R.: Resource analysis by sup-interpretation. In: Hagiya, M., Wadler, P. (eds.) FLOPS 2006. LNCS, vol. 3945, Springer, Heidelberg (2006)
20. Navas, J., Mera, E., López-García, P., Hermenegildo, M.: User-definable resource bounds analysis for logic programs. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, Springer, Heidelberg (2007)
21. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, Springer, Heidelberg (2004)
22. Rosendahl, M.: Automatic Complexity Analysis. In: FPCA, ACM Press, New York (1989)
23. Sands, D.: A naïve time analysis and its theory of cost equivalence. Journal of Logic and Computation 5(4) (1995)
24. Wadler, P.: Strictness analysis aids time analysis. In: POPL (1988)
25. Wegbreit, B.: Mechanical Program Analysis. Comm. of the ACM 18(9) (1975)

SLR: Path-Sensitive Analysis through Infeasible-Path Detection and Syntactic Language Refinement

Gogul Balakrishnan[1], Sriram Sankaranarayanan[1], Franjo Ivančić[1], Ou Wei[2], and Aarti Gupta[1]

[1] NEC Laboratories America, Princeton, NJ, USA
[2] University of Toronto
{bgogul,srirams,ivancic,agupta}@nec-labs.com,
owei@cs.toronto.edu

Abstract. We present a technique for detecting semantically infeasible paths in programs using abstract interpretation. Our technique uses a sequence of path-insensitive forward and backward runs of an abstract interpreter to infer paths in the control flow graph that cannot be exercised in concrete executions of the program.

We then present a syntactic language refinement (SLR) technique that automatically excludes semantically infeasible paths from a program during static analysis. SLR allows us to iteratively prove more properties. Specifically, our technique simulates the effect of a path-sensitive analysis by performing syntactic language refinement over an underlying path-insensitive static analyzer. Finally, we present experimental results to quantify the impact of our technique on an abstract interpreter for C programs.

1 Introduction

Static analysis techniques compute over-approximations of the reachable states for a given program. The theory of abstract interpretation is used to compute such an over-approximation as a fixpoint in a suitably chosen abstract domain [5,6]. The degree of approximation as well as the scalability can be traded-off through a judicious choice of an abstract domain. However, the presence of approximations can cause the analysis to report false positives. Such false positives can be avoided, in part, using techniques such as path-sensitive analysis, disjunctive completion, and various forms of refinements [1,7,8,11,13,17,22].

In practice, many syntactic paths in the control-flow graph (CFG) representation of the program are semantically infeasible, i.e, they may not be traversed by any execution of the program. Reasoning about the infeasibility of such paths is a *key factor* in performing accurate static analyses for checking properties such as correct API usage, absence of null-pointer dereferences and uninitialized use of variables, memory leaks, and so on. Our previous experience with building path-sensitive abstract interpreters [22] also indicates that the benefit of added

© Springer-Verlag Berlin Heidelberg 2008

```
0: if (x > 0)
1:     f := 1;
2: else
3:     f := 0;
4: y := x;
5: if (f > 0)
6:     ASSERT(y >= 0);
```

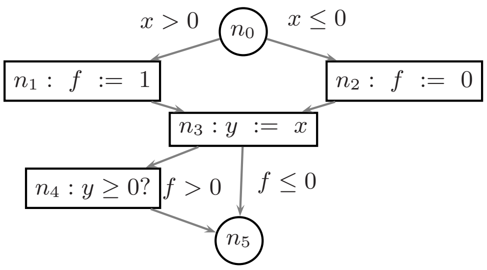


Fig. 1. An example program (left) along with its CFG representation (right). (Node numbers do not correspond to line numbers.)

path sensitivity to static analysis seems to lie mostly in the identification and elimination of semantically infeasible paths.

This paper presents two main contributions. We present a technique based on path-insensitive abstract interpretation to infer semantically infeasible paths in the CFG. Secondly, we use a *syntactic language refinement* scheme for path-sensitive analysis by iteratively removing infeasible paths from the CFG.

Our technique for inferring semantically infeasible paths performs a sequence of many forward and backward runs using a path-insensitive abstract interpreter. We first present infeasibility theorems that use the results of forward/backward fixpoints obtained starting from different initial conditions to characterize paths in the CFG that are semantically infeasible. We then present techniques that enumerate infeasible paths using propositional SAT solvers without repeating previously enumerated paths.

The infeasible paths detected by our technique are excluded from the CFG using syntactic language refinement. Starting with the *syntactic language* defined by the set of all syntactically valid CFG paths, we remove semantically infeasible paths from the syntactic language to obtain *refinement* of the original language. Using a path-insensitive analysis over the refined syntactic language effectively incorporates partial path-sensitivity into the analysis, enabling us to prove properties that are beyond the reach of path-insensitive analyses.

Example 1. The program in Fig. 1 depicts a commonly occurring situation in static analysis. Abstract interpretation using the polyhedral abstract domain is unable to prove the property since it loses the correlation between f and x by computing a join at node n_3. On the other hand, our techniques allow us to prove using path-insensitive analysis that *any semantically feasible path from node n_0 to node n_4 cannot pass through node n_2*. Syntactic language refinement removes this path from the CFG, and performs a new path-insensitive analysis on the remaining paths in the CFG. Such an analysis maintains the correlation between x and f at node n_3 and successfully proves the property.

The F-Soft tool checks C programs for invalid pointer accesses, buffer overflows, memory leaks, incorrect usage of APIs, and arbitrary safety properties specified by a user [14]. We use the techniques described in the paper to improve the path-insensitive analysis used inside the F-Soft tool to obtain the effects of path-sensitive analysis. The resulting analyzer proves more properties with a reasonable resource overhead.

Related Work. Path-sensitive analyses help minimize the impact of the *join* operation at the merge points in the program. A completely path-sensitive analysis is forbiddingly expensive. Many heuristic schemes achieve partial path-sensitive solutions that selectively join or separate the contributions due to different paths using logical disjunctions [8,9,10,13,17,22]. While path-sensitive analysis techniques modify the analysis algorithm, it is possible to achieve path-sensitivity by modifying the abstract domain using powerset extensions [1,16]. Finally, refinement-based techniques can modify the analysis algorithm or the domain itself on demand, based on the failure to prove a property. Gulavani and Rajamani iteratively refine the analysis algorithm by modifying analysis parameters such as widening delays and using weakest preconditions inside a powerset domain [11]. Cousot, Ganty, and Raskin present a fixpoint-based refinement technique that refines an initial Moore-closed abstract domain by adding predicates based on preconditions computed in the concrete domain [7].

Our technique for detecting infeasible paths relies on repeated forward and backward fixpoints. Similar ideas on using path-insensitive analyses to reason about specific program paths have appeared in the context of *abstract debugging* and *semantic slicing* [4,20], which help to interactively zero-in on bugs in code by generating invariants, intermediate assertions, and weakest preconditions.

The use of infeasible paths to refine data-flow analysis has been considered previously by Bodik et al. [3]. However, our approach is more general in many ways: (a) we use abstract interpretation in a systematic manner to handle loops, conditions, procedures, and so on without sacrificing soundness, (b) the underlying analysis used to detect infeasible paths in our approach is itself path-insensitive, which makes it possible to apply our approach on a whole-program basis without requiring much overhead or depth cutoffs.

Ngo and Tan [19] use simple syntactic heuristics to detect infeasible paths in programs that are used in test generation. Surprisingly, their approach seems to detect many infeasible paths using relatively simplistic methods. Such lightweight approaches can also be used as a starting point for syntactic language refinement.

2 Preliminaries

Throughout this paper, we consider single-procedure (while) programs over integer variables. Our results also extend to programs with many procedures and complex datatypes. We use control-flow graphs (CFG) to represent programs. A CFG is a tuple $\langle N, E, V, \mu, n_0, \varphi_0 \rangle$, where N is a set of nodes, $E \subseteq N \times N$ is a set of edges, $n_0 \in N$ is an initial location, V is a set of integer-valued program variables, and φ_0 specifies an initial condition over V that holds at the start of the execution. Each edge $e \in E$ is labeled by a condition or an update $\mu(e)$.

A *state* of the program is a map $s : V \to Z$, specifying the value of each variable. Let Σ be the universe of all valuations to variables in V. A program is assumed to start from the initial location n_0 and a state $s \in [\![\varphi_0]\!]$.

The semantics of an edge $e \in E$ is given by the (concrete) strongest post-condition $\mathsf{post}(e, S)$ and the (concrete) weakest precondition (backward post-condition) $\mathsf{pre}(e, S)$ for sets $S \subseteq \Sigma$. The operator $\mathsf{post} : E \times 2^\Sigma \to 2^\Sigma$ yields the *smallest* set of states reachable upon executing an edge from a given set of states S, while the $\mathsf{pre} : E \times 2^\Sigma \to 2^\Sigma$ yields the *largest* set T such that $t \in T$ iff $\mathsf{post}(e, \{t\}) \subseteq S$. The pre operator is also the post for the "reverse" semantics for the edge e.

A *flow-sensitive* concrete map $\eta : N \to 2^\Sigma$ associates a set of states with each node in the CFG. We denote $\eta_1 \subseteq \eta_2$ iff $\forall n \in N, \eta_1(n) \subseteq \eta_2(n)$.

Forward Propagation.[1] The forward-propagation operator $\mathfrak{F}$ takes a concrete map $\eta : N \to 2^\Sigma$ and returns a new concrete map $\eta' : N \to 2^\Sigma$ such that

$$\eta'(m) = \begin{cases} \bigcup_{\ell \to m \in E} \mathsf{post}(\ell \to m, \eta(\ell)) & \text{if } m \neq n_0 \\ \eta(m) \cup \bigcup_{\ell \to m \in E} \mathsf{post}(\ell \to m, \eta(\ell)) & m = n_0 \end{cases}$$

A map η is *inductive* iff (a) $\eta(n_0) \supseteq [\![\varphi_0]\!]$ and (b) $\eta \supseteq \mathfrak{F}(\eta)$. In other words, if η is inductive, it is also a *post fixpoint* of $\mathfrak{F}$. Since $\mathfrak{F}$ is a monotone operator, the least fixpoint always exists due to Tarski's theorem.

Given a CFG, a *property* consists of a pair $\langle n, \varphi \rangle$ where $n \in N$ is a node, and φ is a first-order assertion representing a set of states. Associated with each property $\langle n, \varphi \rangle$, the CFG has a node $n_E \in N$ and an edge $(n \to n_E) \in E$ with condition φ_E, where φ_E is $\neg\varphi$. The pair $\langle n_E, \varphi_E \rangle$ is referred to as the *error configuration* for property $\langle n, \varphi \rangle$. A property $\langle n, \varphi \rangle$ is verified if the error configuration $\langle n_E, \varphi_E \rangle$ is unreachable, i.e., if $\eta(n_E) = \emptyset$ for an inductive map η.

The least fixpoint of a monotone operator $\mathfrak{F}$ can be computed using *Tarski* iteration. Starting from the initial map η^0, such that $\eta^0(n_0) = [\![\varphi_0]\!]$ and $\eta^0(m) = \emptyset$ for all $m \neq n_0$, we iteratively compute $\eta^{i+1} = \mathfrak{F}(\eta^i)$ until a fixpoint is reached. Unfortunately, this process is computationally infeasible, especially if the program is *infinite state*.

To analyze programs tractably, *abstract interpretation* is used to compute post fixpoints efficiently. An *abstract domain* consists of a lattice $\langle L, \sqsubseteq, \sqcup, \sqcap \rangle$, along with the abstraction map $\alpha : 2^\Sigma \to L$ and the concretization map $\gamma : L \to 2^\Sigma$. Each abstract object $a \in L$ is associated with a set of states $\gamma(a) \subseteq \Sigma$. The maps α and γ together provide a *Galois Connection* between the concrete lattice 2^Σ and the abstract lattice L. The abstract counterparts for the union ($\cup$) and intersection ($\cap$) are the lattice join ($\sqcup$) and lattice meet ($\sqcap$) operators, respectively. Finally, the concrete post-conditions and concrete preconditions have the abstract counterparts post_L and pre_L in the abstract lattice L. A flow-sensitive abstract map $\eta^\sharp : N \to L$ associates each node $n \in N$ to an abstract object $\eta^\sharp(n) \in L$. As before, $\eta_1^\sharp \sqsubseteq \eta_2^\sharp$ iff $\forall n \in N, \eta_1^\sharp(n) \sqsubseteq \eta_2^\sharp(n)$.

[1] Our presentation assumes that the initial node n_0 may have predecessors.

The forward propagation operator $\mathfrak{F}$ can be generalized to a monotone operator $\mathfrak{F}_L : \eta^\sharp \mapsto \eta^{\sharp'}$ in the lattice L such that:

$$\eta^{\sharp'}(m) = \begin{cases} \bigsqcup_{\ell \to m \in E} \mathsf{post}_L(\ell \to m, \eta^\sharp(\ell)) & \text{if } m \neq n_0 \\ \eta^\sharp(m) \sqcup \bigsqcup_{\ell \to m \in E} \mathsf{post}_L(\ell \to m, \eta^\sharp(\ell)) & m = n_0 \end{cases}$$

For a given program, abstract interpretation starts with the initial map $\eta_0^\sharp$, where $\eta_0^\sharp(n_0) = \alpha([\![\varphi_0]\!])$ and $\eta_0^\sharp(m) = \bot$ for all $m \neq n_0$. The process converges to a fixpoint $\eta_F^\sharp$ in L if $\eta_{i+1}^\sharp \sqsubseteq \eta_i^\sharp$. Furthermore, its concretization $\gamma \circ \eta_F^\sharp$ is inductive (post fixpoint) on the concrete lattice. In practice, widening heuristics enforce convergence of the iteration in lattices that do not satisfy the ascending chain condition.

Backward Propagation. An alternative to verifying a property with error configuration $\langle n_E, \varphi_E \rangle$ is *backward propagation* using the backward propagation operator $\mathcal{B}$, which takes a map $\eta : N \to 2^\Sigma$ and returns a new map $\eta' : N \to 2^\Sigma$:

$$\eta'(\ell) = \begin{cases} \eta(\ell) \bigcup_{\ell \to m \in E} \mathsf{pre}(\ell \to m, \eta(m)) & \mathsf{if}\ \ell = n_E \\ \bigcup_{\ell \to m \in E} \mathsf{pre}(\ell \to m, \eta(m)) & \mathsf{otherwise} \end{cases}$$

For an error configuration $\langle n_E, \varphi_E \rangle$, we compute the least fixpoint of $\mathcal{B}$ starting with the initial map η^0 such that $\eta^0(n_E) = [\![\varphi_E]\!]$ and $\eta^0(m) = \emptyset$ for all $m \neq n_E$. A map η is a post fixpoint of the operator $\mathcal{B}$, if $\mathcal{B}(\eta) \subseteq \eta$. The property can be verified if $\eta(n_0) \cap [\![\varphi_0]\!] = \emptyset$, which establishes that it is not possible to reach an error state in $[\![\varphi_E]\!]$ at node n_E from a state in $[\![\varphi_0]\!]$ at the initial node n_0.

Analogous to forward propagation, one may compute a backward (post) fixpoint map $\eta_B^\sharp$ in an abstract domain L by extending the operator $\mathcal{B}$ to the lattice L using the precondition map pre_L to yield $\mathcal{B}_L$. The fixpoint map $\eta_B^\sharp$ computed using $\mathcal{B}_L$ can be used to verify properties.

3 Infeasible-Path Detection

We now characterize infeasible paths in the program using abstract interpretation. Rather than focus on individual paths (of which there may be infinitely many), our results characterize sets of infeasible paths, succinctly. For the remainder of the section, we assume a given abstract domain $\langle L, \sqsubseteq, \sqcup, \sqcap \rangle$ (or even a combination of many abstract domains) that defines the forward operator $\mathfrak{F}_L$ and the backward operator $\mathcal{B}_L$. These operators transform initial maps $\eta_{F_0}^\sharp$ ($\eta_{B_0}^\sharp$) into post fixpoints $\eta_F^\sharp$ ($\eta_B^\sharp$) using abstract forward (backward) propagation. The fixpoint maps $\eta_F^\sharp$ and $\eta_B^\sharp$ are concretized to yield maps $\eta_F = \gamma \circ \eta_F^\sharp$ and $\eta_B = \gamma \circ \eta_B^\sharp$, respectively. Therefore, we present our results in the concrete domain based on concretized fixpoint maps η_F and η_B.

Consider a node $n \in N$ and a set $[\![\varphi]\!]$. We define a basic primitive called *state-set* projection that projects $\langle n, \varphi \rangle$ onto another node $m \in N$ in the CFG as follows: (a) We compute the forward fixpoint map η_F and the backward fixpoint map η_B starting from the following initial map:

$$\eta_0^{\langle n,\varphi\rangle}(\ell) = \begin{cases} [\![\varphi]\!], & \ell = n \\ \bot, & \mathsf{otherwise} \end{cases}$$

(b) The set $\eta_F(m)$ is a *forward projection* and $\eta_B(m)$ is a *backward projection* of $\langle n, \varphi\rangle$ onto m.

Definition 1 (State-set Projection). *A* forward projection *of the pair $\langle n, \varphi\rangle$ onto a node m, denoted $(\langle n, \varphi\rangle \overset{L}{\hookrightarrow} m)$ is the set $\eta_F(m)$, where η_F is a forward (post) fixpoint map starting from the initial map $\eta_0^{\langle n,\varphi\rangle}$.*

Similarly, a backward projection *of the pair $\langle n, \varphi\rangle$ back onto m, denoted $(m \overset{L}{\hookleftarrow} \langle n, \varphi\rangle)$ is the set $\eta_B(m)$, where η_B is the backward fixpoint starting from the initial map $\eta_0^{\langle n,\varphi\rangle}$.*

Forward and backward state-set projections are not unique. They vary, depending on the specific abstract interpretation scheme used to compute them. The projection of a node n onto itself yields the assertion *true*.

Lemma 1. *Let $\varphi_F : \langle n, \varphi\rangle \overset{L}{\hookrightarrow} m$ and $\varphi_B : m \overset{L}{\hookleftarrow} \langle n, \varphi\rangle$ denote the forward and backward projections, respectively, of the pair $\langle n, \varphi\rangle$ onto m. The following hold for state-set projections:*

(1) If an execution starting from a state $s \in [\![\varphi]\!]$ at node n reaches node m with state t, then $t \in [\![\varphi_F]\!]$.
(2) If an execution starting from node m with state t reaches node n with state $s \in [\![\varphi]\!]$ then $t \in [\![\varphi_B]\!]$. $\boxtimes$

3.1 Infeasibility-Type Theorems

The state-set projections computed using forward and backward propagation can be used to detect semantically infeasible paths in a CFG. Let $n_1, \ldots, n_k$ be a subset of nodes in the CFG, n_0 be the initial node and n_{k+1} be some target node of interest. We wish to find if an execution may reach n_{k+1} starting from n_0, while passing through each of the nodes $n_1, \ldots, n_k$, possibly more than once and in an arbitrary order. Let $\Pi(n_0, \ldots, n_{k+1})$ denote the set of all such syntactically valid paths in the CFG.

Let $\varphi_i : \langle n_i, \mathit{true}\rangle \overset{L}{\hookrightarrow} n_{k+1}$, $i \in [0, k+1]$, denote the forward state-set projections from $\langle n_i, \mathit{true}\rangle$ onto the final node n_{k+1}. Similarly, let $\psi_i : n_0 \overset{L}{\hookleftarrow} \langle n_i, \mathit{true}\rangle$, $i \in [0, k+1]$, denote the backward projections from $\langle n_i, \mathit{true}\rangle$ onto node n_0.

Theorem 1 (Infeasibility-type theorem). *The paths in $\Pi(n_0, \ldots, n_{k+1})$ are all semantically infeasible if either*

1. $\varphi_0 \wedge \varphi_1 \wedge \cdots \wedge \varphi_k \wedge \varphi_{k+1} \equiv \emptyset$, or
2. $\psi_0 \wedge \psi_1 \wedge \cdots \wedge \psi_k \wedge \psi_{k+1} \equiv \emptyset$.

Proof. The proof uses facts about the forward and backward state-set projections. From Lem. 1, we conclude that any path that reaches n_{k+1} starting from $\langle n_i, \mathit{true}\rangle$ must do so with a state that satisfies φ_i. As a result, consider a path that traverses all of $n_0, \ldots, n_k$ to reach n_{k+1}. The state at node n_{k+1} must simultaneously satisfy $\varphi_0, \varphi_1, \ldots, \varphi_k$. Since the conjunction of these assertions is empty, we conclude that no such path may exist.

A similar reasoning applies to backward projection from node n_i to node n_0 using the assertions $\psi_0, \ldots, \psi_{k+1}$. ⊠

It is also possible to formulate other infeasibility-type theorems that are similar to Thm. 1 using state-set projection. Let η_F be the forward fixpoint map computed starting from $\langle n_0, \varphi_0\rangle$. Let $\psi_i : \; n_0 \overset{L}{\hookleftarrow} \langle n_i, \eta_F(n_i)\rangle$ be the state-set projection of the set $\eta_F(n_i)$ from node n_i onto node n_0.

Lemma 2. *If $\psi_1 \wedge \ \ldots \wedge \psi_k \ \wedge \ \psi_{k+1} \equiv \emptyset$ then there is no semantically valid path from node n_0 to node n_{k+1} that passes through all of $n_1, \ldots, n_k$.* ⊠

A similar result can be stated for a pair of nodes using the forward and the backward fixpoint maps. Let $\langle n_E, \varphi_E\rangle$ be an error configuration of interest, η_F be the forward (post) fixpoint map computed starting from $\langle n_0, \varphi_0\rangle$ and η_B be the backward (post) fixpoint map computed starting from $\langle n_E, \varphi_E\rangle$.

Lemma 3. *Any error trace that leads to a state in the error configuration $\langle n_E, \varphi_E\rangle$ cannot visit node n' if $\eta_F(n') \wedge \eta_B(n') \equiv \emptyset$.* ⊠

Example 2. Consider the example program shown in Fig. 1. We wish to prove the infeasibility of any path that simultaneously visits n_0, n_2 and n_4. Using state-set projections, we obtain

$$\begin{aligned} \psi_2 :\; n_0 \overset{L}{\hookleftarrow} \langle n_2, \mathit{true}\rangle &= \{x \mid x \leq 0\} \\ \psi_4 :\; n_0 \overset{L}{\hookleftarrow} \langle n_4, \mathit{true}\rangle &= \{x \mid x > 0\} \end{aligned}$$

Since ψ_2 and ψ_4 are disjoint, it is not possible for an execution of the CFG to visit simultaneously the nodes n_0, n_2 and n_4. Likewise, a semantically valid path cannot visit n_2 and n_4 simultaneously, since $\varphi_2 : \; \langle n_2, \mathit{true}\rangle \overset{L}{\hookrightarrow} n_4 \equiv \emptyset$. ⊠

3.2 Infeasible-Path Enumeration

Thus far, we can detect if all the program paths in $\Pi(n_0, \ldots, n_{k+1})$ are semantically infeasible for a given set $n_1, \ldots, n_k, n_{k+1}$. We now consider the problem of *enumerating* such sets using the results derived in the previous sections. Let $N = \{n_0, n_1, \ldots, n_m\}$ denote the set of all nodes in the CFG. In order to apply infeasibility-type theorems such as Thm. 1 and Lem. 2, we compute $m+1$ state-set projections $\psi_0, \ldots, \psi_m$ corresponding to the nodes $n_0, \ldots, n_m$, respectively. Furthermore, to test the subset $\{n_{i_1}, \ldots, n_{i_k}\} \subseteq N$, we test the conjunction $\psi_{i_1} \wedge \cdots \wedge \psi_{i_k}$ for satisfiability. Therefore, to *enumerate* all such subsets, we need to enumerate all index sets $I \subseteq \{1, \ldots, m\}$ such that $\bigwedge_{i \in I} \psi_i \equiv \mathit{false}$. For each such set I, the corresponding subset of N characterizes the infeasible paths.

Definition 2 (Infeasible & Saturated Index Set). *Given assertions $\varphi_1, \ldots, \varphi_m$, an index set $I \subseteq \{1, \ldots, m\}$ is said to be* infeasible *iff $\bigwedge_{j \in I} \varphi_j \equiv false$. Likewise, an infeasible index set I is said to be* saturated *iff no proper subset is itself infeasible.*

Note that each infeasible set is an *unsatisfiable core* of the assertion $\varphi_1 \wedge \cdots \wedge \varphi_m$. Each saturated infeasible set is a *minimal* unsatisfiable core w.r.t set inclusion. Given assertions $\varphi_1, \ldots, \varphi_m$, we wish to enumerate all saturated infeasible index sets. To solve this problem, we provide a generic method using a SAT solver to aid in the enumeration. This method may be improved by encoding the graph structure as a part of the SAT problem to enumerate *continuous* segments in the CFG. We also present domain-specific techniques for directly enumerating all the cores without using SAT solvers.

Generic Enumeration Technique. Assume an oracle O to determine if a conjunctive theory formula $\psi_I : \bigwedge_{i \in I} \psi_i$ is satisfiable. Given O, if ψ_I is unsatisfiable we may extract a minimal unsatisfiable core set $J \subseteq I$ as follows: (1) set $J = I$, (2) for some $j \in J$, if $\psi_{J-\{j\}}$ is unsatisfiable, $J := J - \{j\}$, and (3) repeat step 2 until no more conjuncts need to be considered. Alternatively, O may itself be able to provide a minimal core.

Our procedure maintains a family of subsets $\mathfrak{I} \subseteq 2^{\{1,\ldots,m\}}$ that have not (yet) been checked for feasibility. Starting from $\mathfrak{I} = 2^{\{1,\ldots,m\}}$, we carry out steps (a) and (b), below, until $\mathfrak{I} = \emptyset$:

(a) Pick an untested subset $J \in \mathfrak{I}$.
(b) Check the satisfiability of $\psi_J : \bigwedge_{j \in J} \varphi_j$.
If ψ_J is *satisfiable*, then remove all subsets of J from $\mathfrak{I}$: $\mathfrak{I}' = \mathfrak{I} - \{K \mid K \subseteq J\}$. If ψ_J is *unsatisfiable*, compute a minimal core set $C \subseteq J$. Remove all supersets of C: $\mathfrak{I}' = \mathfrak{I} - \{I | I \supseteq C\}$. Also, *output* C as an infeasible set.

Symbolic enumeration using SAT. In practice, the set $\mathfrak{I}$ may be too large to maintain explicitly. It is therefore convenient to encode it succinctly in a SAT formula. We introduce Boolean *selector variables* $y_1, \ldots, y_m$, where y_i denotes the presence of the assertion φ_i in the theory formula. The set $\mathfrak{I}$ is represented succinctly by a Boolean formula $\mathfrak{F}$ over the selector variables. The initial formula $\mathfrak{F}_0$ is set to *true*. At each step, we may eliminate all supersets of a set J by adding the new clause $\bigvee_{j \in J} \neg y_j$. Fig. 2 shows the procedure to enumerate all infeasible indices using SAT solvers and elimination of unsatisfiable cores.

Graph-based enumeration using SAT. To further improve the enumeration procedure, we note that many subsets $I \subseteq N$ may not lead to any syntactically valid paths through the CFG that visit all nodes in I. Such subsets need not be considered. Nodes n_i and n_j are said to *conflict* if there is no syntactic path starting from n_0 that visits both n_i and n_j. Let $C \subseteq N \times N$ denote the set of all conflicting node pairs. We exclude conflicting nodes or their supersets from the enumeration process by adding the clause $\neg y_i \vee \neg y_j$ for each conflict pair $(n_i, n_j) \in C$. However, in spite of adding the conflict information, syntactically meaningless subsets may still be enumerated by our technique.

```
proc $GenericSATEnumerateCore(\varphi_1, \ldots, \varphi_m)$
  $\mathfrak{F}$ := $true$.
  Add all syntactic constraints to $\mathfrak{F}$.
  while ($\mathfrak{F}$ is satisfiable ) do
    $\langle y_1, \ldots, y_m \rangle$ := satisfying assignment to $\mathfrak{F}$,
    Let $\psi \equiv \bigwedge_{y_i:true} \varphi_i$.
    if $\psi$ is theory satisfiable then
      $\mathfrak{F}$ := $\mathfrak{F} \wedge \bigvee_{y_i:false} y_i$.
    else
      Let $J$ be the unsat core set for $\psi$.
      Output $J$ as an infeasible set.
      $\mathfrak{F}$ := $\mathfrak{F} \wedge \bigvee_{j \in J} \neg y_j$.
    end if
  end while
end proc
```

Fig. 2. SAT-based enumeration modulo theory to enumerate infeasible index sets

Example 3. Consider the CFG skeleton in Fig. 1, disregarding the actual operations in its nodes and edges. We suppose that all paths between nodes n_0 and n_5 are found to be infeasible: i.e., $\{0, 5\}$ is an infeasible index set. Clearly, due to the structure of the CFG, there is now no need to check the satisfiability of the index set $\{0, 3\}$, since all paths to node n_5 have to pass through node n_3. However, this information is not available to the SAT solver, which generates the candidate index set $\{0, 3\}$. $\boxtimes$.

To avoid such paths, we restrict the SAT solver to enumerate *continuous segments* in the CFG.

Definition 3 (Continuous Segment). *A subset $I \subseteq N$ is a* continuous segment *iff for each $n_i \in I$, some successor of n_i (if n_i has any successors) and some predecessor (if n_i has any predecessors) belong to I.*

An infeasible index set I is *syntactically meaningful* iff there exists a continuous segment C such that $I \subseteq C$. Therefore, it suffices to restrict our SAT solver to enumerate all feasible continuous segments C in the CFG. The unsatisfiable core set for any such segment is also a syntactically meaningful set. Secondly, if a continuous segment C is shown to be semantically infeasible by a subset I, we are not interested in other infeasible subsets I' that show C to be infeasible.

Let $p_1, \ldots, p_m$ be the indices of predecessors of a node n_i ($m \geq 1$), and $s_1, \ldots, s_r$ denote the successors indices ($r \geq 1$). We encode continuous segments by adding the following constraints, corresponding to each node n_i in the CFG:

– Forward: If $m > 0$, add $\neg y_i \vee y_{p_1} \vee y_{p_2} \vee \ldots \vee y_{p_m}$.
– Backward: If $r > 0$, add $\neg y_i \vee y_{s_1} \vee y_{s_2} \vee \ldots \vee y_{s_r}$.

The total size of these constraints is linear in the size of the CFG (number of nodes, number of edges). Fig. 2 is modified to add the continuous segment constraints to $\mathfrak{F}$, in addition to the conflicts.

Example 4. Again considering the CFG skeleton from Fig. 1, if the index set $\{0, 5\}$ is found for the unsatisfiable core of the continuous segment $C : \{0, \ldots, 5\}$, the additional clause $\neg y_0 \vee \neg y_5$ will prevent any future consideration of this segment. The index set $\{0, 3\}$ will not be considered, because, n_3 is also part of the continuous segment C which has already been shown to be infeasible.

Utilizing SMT and MAX-SAT Techniques. In principle, a tighter integration of the propositional and theory part can be obtained by enumerating all minimal unsatisfiable cores of an SMT (Satisfiability Modulo Theories) formula $\Psi_m : \wedge_{i=0}^{m}((\neg y_i \vee \varphi_i) \wedge (y_i \vee \neg\varphi_i))$, along with other propositional clauses over $y_1, y_2, \ldots, y_m$ arising from conflict pairs and syntactic graph-based constraints discussed earlier.

The problem of finding all minimal unsatisfiable cores is related to the problem of finding all maximal satisfiable solutions[2] [2,15]. This duality has been exploited to use procedures for finding maximal satisfiable solutions (MAX-SAT) for generating all minimal unsatisfiable cores. This could lead to improved performance, since checking satisfiability is usually considered easier in practice than checking unsatisfiability.

Enumerating Unsatisfiable Cores Directly. In some domains, it is possible to directly enumerate all the unsatisfiable cores of the conjunction $\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_m$. Each unsatisfiable core directly yields infeasible index sets. The advantage of this enumeration method is that it avoids considering index sets for which the corresponding conjunctions is *theory satisfiable*. Secondly, the properties of the underlying abstract domain can be exploited for efficient enumeration. The disadvantage is that the same infeasible index sets may be repeatedly obtained for different unsatisfiable cores.

As a case in point, we consider the interval domain. The concretizations of interval domain objects are conjunctions of the form $x_i \in [l_i, u_i]$. Let $\varphi_1, \ldots, \varphi_m$ be the result of the state projections carried out using interval analysis. We assume that each φ_i is satisfiable. Let φ_i be the assertion $\bigwedge_j x_j \in [l_{ij}, u_{ij}]$, wherein each $l_{ij} \leq u_{ij}$. The lack of relational information in the interval domain restricts each unsatisfiable core to be of size at most 2:

Lemma 4. *Any unsatisfiable core in $\bigwedge_i \varphi_i$ involves exactly two conjuncts: $l_{ij} \leq x_j \leq u_{ij}$ in φ_i and $l_{kj} \leq x_j \leq u_{kj}$ such that $[l_{ij}, u_{ij}] \cap [l_{kj}, u_{kj}] = \emptyset$.* ⊠

As a result, it is more efficient to enumerate infeasible paths using domain-specific reasoning for the interval domain. Direct enumeration of unsatisfiable cores is possible in other domains also. For instance, we may enumerate all the negative cycles for the octagon domain, or all dual polyhedral vertices in the polyhedral domain.

[2] Specifically, the set of all minimal unsatisfiable cores, also called MUS-es, is equivalent to the set of all irreducible hitting sets of all MCS-es, where each MCS is the complement of a maximal satisfiable solution.

```
1: if (x < 0)
2:     x := 1;
3:     y := 1;
4: else
5:     x := 2;
6:     y := 3;
7: if(y = 2)
8:     x := x - 2;
9: ASSERT(x >= 0);
```

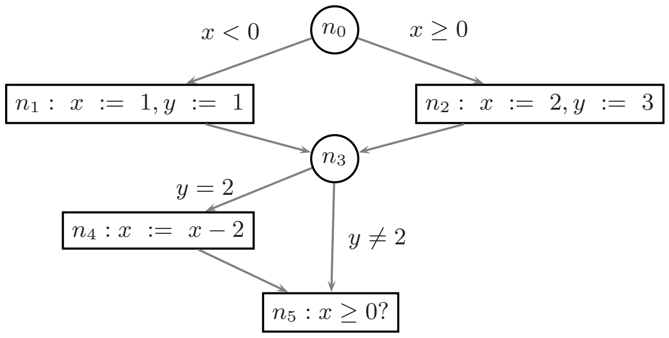


Fig. 3. An example program (left) along with its CFG representation (right). (Node numbers do not correspond to line numbers.)

4 Path-Sensitive Analysis

In this section, we describe how information about infeasible paths discovered in §3 can be used to improve the accuracy of a path-insensitive analysis.

Example 5. Consider the program shown in Fig. 3. Clearly, the assertion at line 9 in the program is never violated. However, neither a forward propagation nor a backward propagation using the interval domain is able to prove the assertion in node n_5. A SAT-based infeasible path enumeration using the interval domain enumerates the sets $\{n_2, n_4\}$ and $\{n_1, n_4\}$ as infeasible. ⊠

Syntactic Language Refinement (SLR). Let $\Pi : \langle N, E, \mu, n_0, \varphi_0 \rangle$ be a CFG with a property $\langle n, \mathit{false} \rangle$ to be established. The *syntactic language* of Π consists of all the edge sequences $e_1, \ldots, e_m$ that may be visited along some walk through the CFG starting from n_0. In effect, we treat Π as an automaton over the alphabet E, wherein each edge e_i accepts the alphabet symbol e_i. Let L_Π denote the language of all edge sequences accepted by CFG Π, represented as a deterministic finite automaton.

The results of the previous section allow us to infer sets $I : \{n_1, \ldots, n_k\} \subseteq N$ such that no semantically valid path from node n_0 to node n may pass through all the nodes of the set I. For each set I, we remove all the (semantically invalid) paths in the set $\Pi(I \cup \{n_0, n\})$ from the syntactic language of the CFG:

$$L'_\Pi = L_\Pi - \underbrace{\{\pi | \pi \text{ is a path from } n_0 \text{ to } n, \text{ passing through all nodes in } I\}}_{L_I}$$

Since the sets L_Π and L_I are regular, $L_{\Pi'} = L_\Pi - L_I$ is also regular. The refinement procedure continues to hold even if L_Π is context free, which happens in the presence of function calls and returns in the CFG.

Let Π' be the automaton recognizing $L_{\Pi'}$. The automaton Π' can be viewed as a CFG once more by associating actions (conditions and assignments) with its

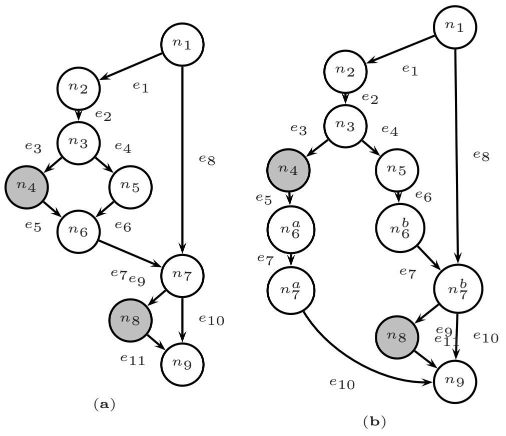


Fig. 4. Path-sensitivity can be simulated by syntactic language refinement: (a) original CFG, (b) paths visiting nodes $\{n_4, n_8\}$ are found infeasible and removed

edges. Each edge labeled with the alphabet e_i is provided the action $\mu(e_i)$ from the original CFG Π. Since $L_{\Pi'} \subseteq L_\Pi$, Π' encodes a smaller set of syntactically valid paths. Therefore, abstract interpretation on Π' may result in a more precise fixpoint.

Example 6. Consider the CFG skeleton Π in Fig. 4(a). Suppose nodes $\{n_4, n_8\}$ were found to be infeasible. The refined CFG Π' is shown in Fig. 4(b). The edges and the nodes are labeled to show the correspondences between the two CFGs. Notice that it is no longer possible to visit the shaded nodes in the same syntactic path.

A path sensitive static analysis *might* be able to obtain the same effect by not using join at node n_6, and partially merging two disjuncts at node n_7. Borrowing the terminology from our previous work the CFG Π' is an *elaboration* of the CFG Π [22], or from the terminology of Mauborgne et al., a trace partitioning [13,17], that keeps the trace visiting node n_4 separate from the other traces. However, our prior knowledge of the infeasibility of $\{n_4, n_8\}$ enables us to automatically rule out the edge $n_7^a \to n_8$. ⊠

From the discussion above, it is evident that syntactic language refinement can be cast in the framework of related schemes such as *elaborations* or *trace-partitions*. However, the key difference is that our scheme always uses the infeasible CFG paths as a partitioning heuristic. The heuristic used in the elaboration or trace-partitioning may be unable to guess the right partitions required to detect the infeasible paths in practice.

Example 7. Returning to the example in Fig. 3, we find that the paths from $n_0 \leadsto n_5$, traversing the nodes $I_0 : \{n_1, n_4\}$ and $I_1 : \{n_2, n_4\}$ are semantically infeasible. Therefore, we may remove such paths from the CFG using syntactic language refinement. The resulting CFG Π' is simply the original CFG with the node n_4 removed. A *path-insensitive* analysis over Π' proves the property. ⊠

```
proc VerifyProperty($\langle n, \varphi \rangle$)
  $I$ := $N$
  Let $\langle n_E, \varphi_E \rangle$ be the error configuration for property $\langle n, \varphi \rangle$
  while ( $I \neq \emptyset$ ) do
    Let $\eta_F$ be the forward fixpoint computed starting from $\langle n_0, \varphi_0 \rangle$
    Let $\eta_B$ be the backward fixpoint computed starting from $\langle n_E, \varphi_E \rangle$
    $I$ := $\emptyset$
    for all conditional branches $\ell \rightarrow m \in E$ do
      if ( $\eta_F(m) \sqcap \eta_B(m) \equiv \mathit{false}$) then
        Let $I$ := $I \cup \{n \in N \mid n$ is control dependent on $\ell \rightarrow m\}$.
      end if
    end for
    Remove the nodes in $I$ from the CFG
    if $\eta_F(n) \equiv \mathit{false}$ then
      return PROVED.
    end if
  end while
  return NOT PROVED.
end proc
```

Fig. 5. Using infeasible-path detection to improve path-insensitive analysis

In theory, it is possible to first remove infeasible path segments using abstract interpretation, perform a language refinement, and subsequently, analyze the refined CFG. In practice, however, we observe that most infeasible paths involve no more than two intermediate nodes. Furthermore, the size of the refined CFG after a set I has been removed can be a factor $2^{|I|}$ larger.

Application. We now present a simple version of the SLR scheme using Lem. 3 that removes at most one intermediate node w.r.t a given property. Secondly, we repeatedly refine the CFG at each stage by using the improved abstract interpretation result on the original CFG. Finally, thanks to the form of Lem. 3, each application of the scheme requires just two fixpoint computations, one in the forward direction and the other in the backward.

Fig. 5 shows an iterative syntactic language refinement scheme. Each step involves a forward fixpoint from the initial node and a backward fixpoint computed from the property node. First, infeasible pairs of nodes are then determined using Lem. 3, and the paths involving such pairs are pruned from the CFG. Since paths are removed from the CFG, subsequent iterations can produce stronger fixpoints, and therefore, detect more infeasible intermediate nodes. The language refinement is repeated until the property is verified, or no new nodes are detected as infeasible in consecutive iterations.

Example 8. VerifyProperty proves the assertion in the example shown in Fig. 3. During the first iteration, the condition at line 8 of *VerifyProperty* holds for the edge $n_0 \rightarrow n_2$. Consequently, node n_2 will be removed before the next forward fixpoint computation. Hence, interval analysis will be able to determine that edge $n_3 \rightarrow n_4$ is infeasible, and therefore, the property $\langle n_5, x \geq 0 \rangle$ is verified. ⊠

5 Experiments

We have implemented the SLR technique inside the F-SOFT C program verifier [14] to check array, pointer, and C string usage. The analyzer is *context sensitive*, by using call strings to track contexts. Our abstract interpreter supports a combination of different numerical domains, including constant folding, interval, octagon [18], polyhedron [12] and the symbolic range domains [21]. The experiments were run on a Linux machine with two Xeon 2.8GHz processors and 4GB of memory.

Infeasible-Path Enumeration. We implemented the algorithm in Fig. 2 using the octagon abstract domain as a proof-of-concept. Tab. 1 shows the performance of the infeasible-path enumerator over a set of small but complex functions written in C [23]. As an optimization, we modified the algorithm to enumerate infeasible sets solely involving conditional branches. The running time of the algorithm is a function of the number of conditional branches and the number of variables in the program. Surprisingly, computing fixpoints accounts for the majority of the running time. Not surprisingly, almost all saturated infeasible sets involved exactly two edges. With the additional optimizations described here and a variable-packing heuristic, we hope to scale this technique to larger functions.

Syntactic Language Refinement. We implemented the *VerifyProperty* algorithm on top of our existing abstract interpreter. Given a program, we first run a series of path-insensitive analyses. Properties thus proved are sliced away from the CFG. The resulting simplified CFG is fed into our analysis. Our analysis is run twice: first, using the interval domain, and then using the octagon domain on the sliced CFG from the interval analysis instantiation. Tab. 2 shows the performance of our tool chain on a collection of industrial as well as open source projects. For each program, we show the number of proofs obtained and the time taken by the base analysis as well as the *additional proofs* along with the overhead of the SLR technique using the intervals and the octagon domains successively. For our set of examples, the SLR technique obtains 15% more proofs over and above the base analysis. However, it involves a significant time overhead of roughly 15% for the interval domain and 75% for the octagon domain. A preliminary comparison

Table 1. Number of saturated infeasible sets from SAT-based enumeration

				Time (s)		
Prog.	LOC	#Vars	#Branches	FixPoint	Enum	#Inf.Sets
ex1	35	7	13	0.07	0.01	3
ex2	40	6	15	0.08	0.02	10
ex3	79	9	36	1.46	0.12	6
ex4	85	71	41	2160.67	6.12	0
ex5	94	12	40	2.85	3.60	38
ex6	102	38	27	51.76	0.15	2
ex7	115	2	31	0.06	0.02	28

Table 2. Performance of the tool flow using SLR. (H_i: mobile software application modules, L_i: Linux device drivers, M_i: a network protocol implementation modules.)

Code	Simplified LOC	Base analysis		*Additional* Proofs with SLR			
				Intervals		Octagons	
		Proofs	Time (s)	Proofs	Time (s)	Proofs	Time (s)
H_1	2282	7/14	28.08	0/7	0.5	0/7	8.77
H_2	3319	33/49	355.85	0/16	7.04	0/16	102.24
H_3	2668	23/37	52.49	0/14	1.25	0/14	14.17
L_4	4626	12/31	10.21	0/19	5.07	0/19	9.1
M_5	5095	67/265	69.62	35/198	84.99	28/163	217.44
L_6	5346	6/20	3.62	0/14	1.53	0/14	7.59
L_7	5599	5/146	681.48	0/141	198.26	0/141	239.39
M_8	6142	109/314	86.85	1/205	173.74	98/204	2008.07
L_9	6326	119/135	70.88	0/16	2.74	0/16	8.18
M_{10}	6645	93/385	244.57	25/292	390.15	70/267	990.11
M_{11}	7541	162/442	262.94	77/280	361.08	49/203	1069.1
M_{12}	10206	285/745	1479.29	30/460	1584.45	154/430	6681.36
M_{13}	11803	325/786	1089.86	121/461	859.36	99/340	5710.56
L_{14}	13162	38/114	289.9	34/76	130.98	0/42	263.86
M_{15}	14665	313/606	117.96	163/293	112.19	10/130	204.16
M_{17}	26758	1904/1918	2208.85	0/14	0.95	0/14	8.89
M_{18}	47213	4173/4218	16880.00	21/45	4.07	0/24	20.51
Total		7674/10225		507/2551		508/2044	

Table 3. Comparison of SLR with path-sensitive analysis using CFG elaborations [22]. (#T: total time with two outlying data points removed, #P: additional proofs.)

	Base			CFG Elaborations [22]		SLR			
						Cont. Insens.		Cont. Sens.	
#Progs	Tot.	Proofs	#T	#P	#T	#P	#T	#P	#T
48	403	178	2.47	+45	76.42	+44	27	+79	36

with our previous work suggests that this overhead is quite competitive with related techniques for performing path-sensitive analysis [22].

Tab. 3 shows a direct comparison of our implementation with a partially path-sensitive analysis implemented using CFG elaborations [22]. The comparison is carried out over a collection of small example programs [23] written in the C language. These programs range from 20-400 lines of code, and are designed to evaluate the handling of loops, aliasing, dynamic allocation, type-casts, string library functions and other sources of complexities in practical programs. Because the latter implementation is context-insensitive, we compare against a context-insensitive version of our technique as well. CFG elaboration technique achieves 45 extra proofs with 50x time overhead. Our context-insensitive implementation proves roughly as many properties as CFG elaborations with a much smaller overhead (roughly 10x for context-insensitive and 20x for context-sensitive). Our implementation proved quite a few properties that could not be established by elaborations and vice versa. Not surprisingly, added context-sensitivity to our technique renders it vastly superior.

References

1. Bagnara, R., Hill, P.M., Zaffanella, E.: Widening operators for powerset domains. STTT 9(3-4), 413–414 (2007)
2. Bailey, J., Stuckey, P.J.: Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In: PADL (2005)
3. Bodik, R., Gupta, R., Soffa, M.L.: Refining data flow information using infeasible paths. In: Biham, E. (ed.) FSE 1997. LNCS, vol. 1267, pp. 361–377. Springer, Heidelberg (1997)
4. Bourdoncle, F.: Abstract debugging of higher-order imperative languages. In: PLDI 1993, pp. 46–55. ACM Press, New York (1993)
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In: Proc. Principles of Programming Languages (POPL) (1977)
6. Cousot, P., Cousot, R.: Static determination of dynamic properties of recursive procedures. In: Formal Descriptions of Programming Concepts, North-Holland, Amsterdam (1978)
7. Cousot, P., Ganty, P., Raskin, J.-F.: Fixpoint-guided abstraction refinements. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 333–348. Springer, Heidelberg (2007)
8. Das, M., Lerner, S., Seigle, M.: ESP: path-sensitive program verification in polynomial time. In: PLDI, pp. 57–68. ACM Press, New York (2002)
9. Dhurjati, D., Das, M., Yang, Y.: Path-sensitive dataflow analysis with iterative refinement. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 425–442. Springer, Heidelberg (2006)
10. Fischer, J., Jhala, R., Majumdar, R.: Joining dataflow with predicates. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, Springer, Heidelberg (2005)
11. Gulavani, B.S., Rajamani, S.K.: Counterexample driven refinement for abstract interpretation. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, pp. 474–488. Springer, Heidelberg (2006)
12. Halbwachs, N., Proy, Y.-E., Roumanoff, P.: Verification of real-time systems using linear relation analysis. Formal Methods in Sys. Design 11(2), 157–185 (1997)
13. Handjieva, M., Tzolovski, S.: Refining static analyses by trace-based partitioning using control flow. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, Springer, Heidelberg (1998)
14. Ivančić, F., Shlyakhter, I., Gupta, A., Ganai, M.K., Kahlon, V., Wang, C., Yang, Z.: Model checking C programs using F-SOFT. In: ICCD, pp. 297–308 (2005)
15. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. of Automated Reasoning 40(1), 133 (2008)
16. Manevich, R., Sagiv, S., Ramalingam, G., Field, J.: Partially disjunctive heap abstraction. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 265–279. Springer, Heidelberg (2004)
17. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, Springer, Heidelberg (2005)

18. Miné, A.: The octagon abstract domain. In: Working Conf. on Reverse Eng. (2001)
19. Ngo, M.N., Tan, H.B.K.: Detecting large number of infeasible paths through recognizing their patterns. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, Springer, Heidelberg (2007)
20. Rival, X.: Understanding the origin of alarms in ASTRÉE. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, Springer, Heidelberg (2005)
21. Sankaranarayanan, S., Ivančić, F., Gupta, A.: Program analysis using symbolic ranges. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 366–383. Springer, Heidelberg (2007)
22. Sankaranarayanan, S., Ivančić, F., Shlyakhter, I., Gupta, A.: Static analysis in disjunctive numerical domains. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 3–17. Springer, Heidelberg (2006)
23. Sankaranarayanan, S.: NECLA static analysis benchmarks (2007), http://www.nec-labs.com/~fsoft

Flow Analysis, Linearity, and PTIME

David Van Horn and Harry G. Mairson

Department of Computer Science
Brandeis University
Waltham, Massachusetts 02454
{dvanhorn,mairson}@cs.brandeis.edu

Abstract. Flow analysis is a ubiquitous and much-studied component of compiler technology—and its variations abound. Amongst the most well known is Shivers' 0CFA; however, the best known algorithm for 0CFA requires time cubic in the size of the analyzed program and is unlikely to be improved. Consequently, several analyses have been designed to approximate 0CFA by trading precision for faster computation. Henglein's simple closure analysis, for example, forfeits the notion of directionality in flows and enjoys an "almost linear" time algorithm. But in making trade-offs between precision and complexity, what has been given up and what has been gained? Where do these analyses differ and where do they coincide?

We identify a core language—the linear λ-calculus—where 0CFA, simple closure analysis, and many other known approximations or restrictions to 0CFA are rendered identical. Moreover, for this core language, analysis corresponds with (instrumented) evaluation. Because analysis faithfully captures evaluation, and because the linear λ-calculus is complete for PTIME, we derive PTIME-completeness results for all of these analyses.

1 Introduction

Flow analysis [1,2,3,4] is concerned with providing sound approximations to the question of "does a given value flow into a given program point during evaluation?" The most approximate analysis will always answer *yes*, which takes no resources to compute—and is of little use. On the other hand, the most precise analysis will answer *yes* if and only if the given value flows into the program point during evaluation, which is useful, albeit uncomputable. In mediating between these extremes, every static analysis necessarily gives up valuable information for the sake of computing an answer within bounded resources. Designing a static analyzer, therefore, requires making trade-offs between precision and complexity. But what exactly is the analytic relationship between forfeited information and resource usage for any given design decision? In other words:

What are the computationally potent ingredients in a static analysis?

The best known algorithms to compute Shivers' 0CFA [5], a canonical flow analysis for higher-order programs, are cubic in the size of the program, and there

© Springer-Verlag Berlin Heidelberg 2008

is strong evidence to suggest that in general, this cannot be improved [6]. Nonetheless, information can be given up in the service of quickly computing a necessarily less precise analysis, avoiding the "cubic bottleneck." For example, by forfeiting 0CFA's notion of directionality, algorithms for Henglein's simple closure analysis run in near linear time [7]. Similarly, by explicitly bounding the number of passes the analyzer is allowed over the program, as in Ashley and Dybvig's sub-0CFA [8], we can recover running times that are linear in the size of the program. But the question remains: Can we do better? For example, is it possible to compute these less precise analyses in logarithmic space? We show that without profound combinatorial breakthroughs (PTIME = LOGSPACE), the answer is no. Simple closure analysis, sub-0CFA, and other analyses that approximate or restrict 0CFA, *require*—and are therefore, like 0CFA [9], complete for—polynomial time.

What is flow analysis? Flow analysis is *the* ubiquitous static analysis of higher-order programs. As Heintze and McAllester remark, some form of flow analysis is used in most forms of analysis for higher-order languages [10]. It answers fundamental questions such as *what values can a given subexpression possibly evaluate to at run-time?* Each subexpression is identified with a unique superscripted *label* ℓ, which serves to index that program point. The result of a flow analysis is a *cache* $\widehat{\mathsf{C}}$ that maps program points (and variables) to sets of values. These analyses are *conservative* in the following sense: if v is in $\widehat{\mathsf{C}}(\ell)$, then the subexpression label ℓ *may* evaluate to v when the program is run (likewise, if v is in $\widehat{\mathsf{C}}(x)$, x may be bound to v when the program is run). But if v is *not* in $\widehat{\mathsf{C}}(\ell)$, we know that e *cannot* evaluate to v and conversely if e evaluates to v, v *must* be in $\widehat{\mathsf{C}}(\ell)$.

For the purposes of this paper and all of the analyses considered herein, values are (possibly open) λ-abstractions. During evaluation, functional values are denoted with *closures*—a λ-abstraction together with an *environment* that closes it. Values considered in the analysis approximate run-time denotations in the sense that if a subexpression labeled ℓ evaluates to the closure $\langle \lambda x.e, \rho \rangle$, then $\lambda x.e$ is in $\widehat{\mathsf{C}}(\ell)$.

The *acceptability* of a flow analysis is often specified as a set of (in)equations on program fragments. The most naive way to compute a satisfying analysis is to initialize the cache with the flow sets being empty. Successive passes are then made over the program, monotonically updating the cache as needed, until the least fixed point is reached. The more approximate the analysis, the faster this algorithm converges on a fixed point. The key to a fruitful analysis, then, is "to accelerate the analysis without losing too much information" [8].

One way to realize the computational potency of a static analysis is to subvert this loss of information, making the analysis an *exact* computational tool. Proving lower bounds on the expressiveness of an analysis becomes an exercise in hacking, armed with this newfound tool. Clearly the more approximate the analysis, the less there is to work with, computationally speaking, and the more we have to do to undermine the approximation. But a fundamental technique allows us to understand expressivity in static analysis: *linearity.* This paper serves to demonstrate that linearity renders a number of the most approximate flow analyses both equivalent and exact.

Linearity and approximation in static analysis: Linearity, the Curry-Howard programming counterpart of linear logic [11], plays an important role in understanding static analyses. The reason is straightforward: because static analysis has to be tractable, it typically approximates normalization, instead of simulating it, because running the program may take too long. For example, in the analysis of simple types—surely a kind of static analysis—the approximation is that all occurrences of a bound variable must have the same type (as a consequence, perfectly good programs are rejected). A comparable but not identical thing happens in the case of type inference for ML and bounded-rank intersection types—but note that when the program is linear, there is no approximation, and type inference becomes evaluation under another name.

In the case of flow analysis, similarly, a cache is computed in the course of an approximate evaluation, which is only an approximation because each evaluation of an abstraction body causes the same piece of the cache to be (monotonically) updated. Again, if the term is linear, then there is only one evaluation of the abstraction body, and the flow analysis becomes synonymous with normalization.

Organization of this paper: The next section introduces 0CFA in order to provide intuitions and a point of reference for comparisons with subsequent analyses. Section 3 specifies and provides an algorithm for computing Henglein's simple closure analysis. Section 4 develops a correspondence between evaluation and analysis for linear programs. We show that when the program is linear, normalization and analysis are synonymous. As a consequence the normal form of a program can be *read back* from the analysis. We then show in Section 5 how to simulate circuits, the canonical PTIME-hard problem, using linear terms. This establishes the PTIME-hardness of the analysis. Finally, we discuss other monovariant flow analyses and sketch why these analyses remain complete for PTIME and provide conclusions and perspective.

2 Shivers' 0CFA

As a starting point, we consider Shivers' 0CFA [5,3].[1]

The Language: A countably infinite set of labels, which include variable names, is assumed. The syntax of the language is given by the following grammar:

Exp	$e ::= t^\ell$	expressions (or labeled terms)
Term	$t ::= x \mid e\ e \mid \lambda x.e$	terms (or unlabeled expressions)

All of the syntactic categories are implicitly understood to be restricted to the finite set of terms, labels, variables, etc. that occur in the *program of interest*—the program being analyzed. The set of labels, which includes variable names, in

[1] It should be noted that Shivers' original *zeroth-order control-flow analysis* (0CFA) was developed for a core CPS-Scheme language, whereas the analysis considered here is in direct-style. Sestoft independently developed a similar flow analysis in his work on globalization [2,12]. See [4,13] for details.

a program fragment is denoted $\mathbf{lab}(e)$. As a convention, programs are assumed to have distinct bound variable names.

The result of 0CFA is an *abstract cache* that maps each program point (i.e., label) to a set of λ-abstractions which potentially flow into this program point at run-time:

$$\widehat{\mathsf{C}} \in \mathbf{Lab} \to \mathcal{P}(\mathbf{Term}) \qquad \text{abstract caches}$$

Caches are extended using the notation $\widehat{\mathsf{C}}[\ell \mapsto s]$, and we write $\widehat{\mathsf{C}}[\ell \mapsto^{+} s]$ to mean $\widehat{\mathsf{C}}[\ell \mapsto (s \cup \widehat{\mathsf{C}}(\ell))]$. It is convenient to sometimes think of caches as mutable tables (as we do in the algorithm below), so we abuse syntax, letting this notation mean both functional extension and destructive update. It should be clear from context which is implied.

The Analysis: We present the specification of the analysis here in the style of Nielson *et al.* [14]. Each subexpression is identified with a unique superscripted *label* ℓ, which marks that program point; $\widehat{\mathsf{C}}(\ell)$ stores all possible values flowing to point ℓ. An *acceptable* flow analysis for an expression e is written $\widehat{\mathsf{C}} \models e$:

$$\begin{aligned}
&\widehat{\mathsf{C}} \models x^{\ell} \text{ iff } \widehat{\mathsf{C}}(x) \subseteq \widehat{\mathsf{C}}(\ell)\\
&\widehat{\mathsf{C}} \models (\lambda x.e)^{\ell} \text{ iff } \lambda x.e \in \widehat{\mathsf{C}}(\ell)\\
&\widehat{\mathsf{C}} \models (t_1^{\ell_1}\ t_2^{\ell_2})^{\ell} \text{ iff } \widehat{\mathsf{C}} \models t_1^{\ell_1} \wedge \widehat{\mathsf{C}} \models t_2^{\ell_2} \wedge \forall \lambda x.t_0^{\ell_0} \in \widehat{\mathsf{C}}(\ell_1) :\\
&\qquad\qquad \widehat{\mathsf{C}} \models t_0^{\ell_0} \wedge \widehat{\mathsf{C}}(\ell_2) \subseteq \widehat{\mathsf{C}}(x) \wedge \widehat{\mathsf{C}}(\ell_0) \subseteq \widehat{\mathsf{C}}(\ell)
\end{aligned}$$

The $\models$ relation needs to be coinductively defined since verifying a judgment $\widehat{\mathsf{C}} \models e$ may obligate verification of $\widehat{\mathsf{C}} \models e'$ which in turn may require verification of $\widehat{\mathsf{C}} \models e$. The above specification of acceptability, when read as a table, defines a functional, which is monotonic, has a fixed point, and $\models$ is defined coinductively as the greatest fixed point of this functional.[2]

Writing $\widehat{\mathsf{C}} \models t^{\ell}$ means "the abstract cache $\widehat{\mathsf{C}}$ contains all the flow information for program fragment t at program point ℓ." The goal is to determine the *least* cache solving these constraints to obtain the most precise analysis. Caches are partially ordered with respect to the program of interest:

$$\widehat{\mathsf{C}} \sqsubseteq \widehat{\mathsf{C}}' \text{ iff } \forall \ell : \widehat{\mathsf{C}}(\ell) \subseteq \widehat{\mathsf{C}}'(\ell).$$

Since we are concerned only with the least cache (the most precise analysis) we refer to this as *the* cache, or synonymously, *the* analysis.

The Algorithm: These constraints can be thought of as an abstract evaluator—$\widehat{\mathsf{C}} \models t^{\ell}$ simply means *evaluate* t^{ℓ}, which serves *only* to update an (initially empty) cache.

$$\begin{aligned}
\mathcal{A}[\![x^{\ell}]\!] &= \widehat{\mathsf{C}}[\ell \mapsto^{+} \widehat{\mathsf{C}}(x)]\\
\mathcal{A}[\![(\lambda x.e)^{\ell}]\!] &= \widehat{\mathsf{C}}[\ell \mapsto \{\lambda x.e\}]\\
\mathcal{A}[\![(t_1^{\ell_1}\ t_2^{\ell_2})^{\ell}]\!] &= \mathcal{A}[\![t_1^{\ell_1}]\!];\quad \mathcal{A}[\![t_2^{\ell_2}]\!];\\
&\qquad \textbf{for each } \lambda x.t_0^{\ell_0} \textbf{ in } \widehat{\mathsf{C}}(\ell_1) \textbf{ do}\\
&\qquad\quad \widehat{\mathsf{C}}[x \mapsto^{+} \widehat{\mathsf{C}}(\ell_2)];\quad \mathcal{A}[\![t_0^{\ell_0}]\!];\quad \widehat{\mathsf{C}}[\ell \mapsto^{+} \widehat{\mathsf{C}}(\ell_0)]
\end{aligned}$$

[2] See [14] for a thorough discussion of coinduction in specifying static analyses.

The abstract evaluator $\mathcal{A}[\![\cdot]\!]$ is iterated until the finite cache reaches a fixed point.[3] Since the cache size is polynomial in the program size, so is the running time, as the cache is *monotonic*—we put values in, but never take them out. Thus the analysis and any decision problems answered by the analysis are clearly computable within polynomial time.

An Example: Consider the following program, which we will return to discuss further in subsequent analyses:

$$((\lambda f.((f^1 f^2)^3 (\lambda y.y^4)^5)^6)^7 (\lambda x.x^8)^9)^{10}$$

The 0CFA is given by the following cache:

$$\begin{array}{lll}
\widehat{\mathsf{C}}(1) = \{\lambda x\} & \widehat{\mathsf{C}}(6) \;\, = \{\lambda x, \lambda y\} & \\
\widehat{\mathsf{C}}(2) = \{\lambda x\} & \widehat{\mathsf{C}}(7) \;\, = \{\lambda f\} & \widehat{\mathsf{C}}(f) = \{\lambda x\} \\
\widehat{\mathsf{C}}(3) = \{\lambda x, \lambda y\} & \widehat{\mathsf{C}}(8) \;\, = \{\lambda x, \lambda y\} & \widehat{\mathsf{C}}(x) = \{\lambda x, \lambda y\} \\
\widehat{\mathsf{C}}(4) = \{\lambda y\} & \widehat{\mathsf{C}}(9) \;\, = \{\lambda x\} & \widehat{\mathsf{C}}(y) = \{\lambda y\} \\
\widehat{\mathsf{C}}(5) = \{\lambda y\} & \widehat{\mathsf{C}}(10) = \{\lambda x, \lambda y\} &
\end{array}$$

where we write λx as shorthand for $\lambda x.x^8$, etc.

3 Henglein's Simple Closure Analysis

Simple closure analysis follows from an observation by Henglein some 15 years ago: he noted that the standard flow analysis can be computed in dramatically less time by changing the specification of flow constraints to use equality rather than containment [7]. The analysis bears a strong resemblance to simple typing: analysis can be performed by emitting a system of equality constraints and then solving them using *unification*, which can be computed in almost linear time with a union-find datastructure.

Consider a program with both $(f\ \ x)$ and $(f\ \ y)$ as subexpressions. Under 0CFA, whatever flows into x and y will also flow into the formal parameter of all abstractions flowing into f, but it is not necessarily true that whatever flows into x *also* flows into y and *vice versa*. However, under simple closure analysis, this is the case. For this reason, flows in simple closure analysis are said to be *bidirectional*.

The Analysis:

$$\begin{array}{l}
\widehat{\mathsf{C}} \models x^\ell \text{ iff } \widehat{\mathsf{C}}(x) = \widehat{\mathsf{C}}(\ell) \\
\widehat{\mathsf{C}} \models (\lambda x.e)^\ell \text{ iff } \lambda x.e \in \widehat{\mathsf{C}}(\ell) \\
\widehat{\mathsf{C}} \models (t_1^{\ell_1}\ t_2^{\ell_2})^\ell \text{ iff } \widehat{\mathsf{C}} \models t_1^{\ell_1} \wedge \widehat{\mathsf{C}} \models t_2^{\ell_2} \wedge \forall \lambda x.t_0^{\ell_0} \in \widehat{\mathsf{C}}(\ell_1) : \\
\qquad\qquad \widehat{\mathsf{C}} \models t_0^{\ell_0} \wedge \widehat{\mathsf{C}}(\ell_2) = \widehat{\mathsf{C}}(x) \wedge \widehat{\mathsf{C}}(\ell_0) = \widehat{\mathsf{C}}(\ell)
\end{array}$$

[3] A single iteration of $\mathcal{A}[\![e]\!]$ may in turn make a recursive call $\mathcal{A}[\![e]\!]$ with no change in the cache, so care must be taken to avoid looping. This amounts to appealing to the coinductive hypothesis $\widehat{\mathsf{C}} \models e$ in verifying $\widehat{\mathsf{C}} \models e$. However, we consider this inessential detail, and it can safely be ignored for the purposes of obtaining our main results in which this behavior is never triggered.

The Algorithm: We write $\widehat{\mathsf{C}}[\ell \leftrightarrow \ell']$ to mean $\widehat{\mathsf{C}}[\ell \mapsto^{+} \widehat{\mathsf{C}}(\ell')][\ell' \mapsto^{+} \widehat{\mathsf{C}}(\ell)]$.

$$\begin{aligned}
\mathcal{A}[\![x^{\ell}]\!] &= \widehat{\mathsf{C}}[\ell \leftrightarrow x]\\
\mathcal{A}[\![(\lambda x.e)^{\ell}]\!] &= \widehat{\mathsf{C}}[\ell \mapsto^{+} \{\lambda x.e\}]\\
\mathcal{A}[\![(t_1^{\ell_1} t_2^{\ell_2})^{\ell}]\!] &= \mathcal{A}[\![t_1^{\ell_1}]\!];\quad \mathcal{A}[\![t_2^{\ell_2}]\!];\\
&\quad \textbf{for each } \lambda x.t_0^{\ell_0} \textbf{ in } \widehat{\mathsf{C}}(\ell_1) \textbf{ do}\\
&\quad\quad \widehat{\mathsf{C}}[x \leftrightarrow \ell_2];\quad \mathcal{A}[\![t_0^{\ell_0}]\!];\quad \widehat{\mathsf{C}}[\ell \leftrightarrow \ell_0]
\end{aligned}$$

The abstract evaluator $\mathcal{A}[\![\cdot]\!]$ is iterated until a fixed point is reached.[4] By similar reasoning to that given for 0CFA, simple closure analysis is clearly computable within polynomial time.

An Example: Recall the example program of the previous section:

$$((\lambda f.((f^1 f^2)^3 (\lambda y.y^4)^5)^6)^7 (\lambda x.x^8)^9)^{10}$$

Notice that $\lambda x.x$ is applied to itself and then to $\lambda y.y$, so x will be bound to both $\lambda x.x$ and $\lambda y.y$, which induces an equality between these two terms. Consequently, wherever 0CFA gives a flow set of $\{\lambda x\}$ or $\{\lambda y\}$, simple closure analysis will give $\{\lambda x, \lambda y\}$. The simple closure analysis is given by the following cache (new flows are underlined):

$$\begin{array}{lll}
\widehat{\mathsf{C}}(1) = \{\lambda x, \underline{\lambda y}\} & \widehat{\mathsf{C}}(6) \;\, = \{\lambda x, \lambda y\} & \\
\widehat{\mathsf{C}}(2) = \{\lambda x, \underline{\lambda y}\} & \widehat{\mathsf{C}}(7) \;\, = \{\lambda f\} & \widehat{\mathsf{C}}(f) = \{\lambda x, \underline{\lambda y}\}\\
\widehat{\mathsf{C}}(3) = \{\lambda x, \lambda y\} & \widehat{\mathsf{C}}(8) \;\, = \{\lambda x, \lambda y\} & \widehat{\mathsf{C}}(x) = \{\lambda x, \lambda y\}\\
\widehat{\mathsf{C}}(4) = \{\lambda y, \underline{\lambda x}\} & \widehat{\mathsf{C}}(9) \;\, = \{\lambda x, \underline{\lambda y}\} & \widehat{\mathsf{C}}(y) = \{\lambda y, \underline{\lambda x}\}\\
\widehat{\mathsf{C}}(5) = \{\lambda y, \underline{\lambda x}\} & \widehat{\mathsf{C}}(10) = \{\lambda x, \lambda y\} &
\end{array}$$

4 Linearity and Normalization

In this section, we show that when the program is *linear*—every bound variable occurs exactly once—analysis and normalization are synonymous.

First, consider an evaluator for our language, $\mathcal{E}[\![\cdot]\!]$:

$$\mathcal{E}[\![\cdot]\!] : \textbf{Exp} \rightarrow \textbf{Env} \rightharpoonup \langle \textbf{Term}, \textbf{Env} \rangle$$

$$\begin{aligned}
\mathcal{E}[\![x^{\ell}]\!][x \mapsto c] &= c\\
\mathcal{E}[\![(\lambda x.e)^{\ell}]\!]\rho &= \langle \lambda x.e, \rho \rangle\\
\mathcal{E}[\![(e_1\ e_2)^{\ell}]\!]\rho &= \textbf{let } \langle \lambda x.e_0, \rho' \rangle = \mathcal{E}[\![e_1]\!]\rho \restriction \textbf{fv}(e_1) \textbf{ in}\\
&\quad\ \textbf{let } c = \mathcal{E}[\![e_2]\!]\rho \restriction \textbf{fv}(e_2) \textbf{ in}\\
&\quad\quad \mathcal{E}[\![e_0]\!]\rho'[x \mapsto c]
\end{aligned}$$

We use ρ to range over *environments*, $\textbf{Env} = \textbf{Var} \rightharpoonup \langle \textbf{Term}, \textbf{Env} \rangle$, and let c range over *closures*, each comprising a term and an environment that closes the

[4] The fine print of footnote [3] applies as well.

term. The function $\mathbf{lab}(\cdot)$ is extended to closures and environments by taking the union of all labels in the closure or in the range of the environment, respectively.

Notice that the evaluator "tightens" the environment in the case of an application, thus maintaining throughout evaluation that the domain of the environment is exactly the set of free variables in the expression. When evaluating a variable occurrence, there is only one mapping in the environment: the binding for this variable. Likewise, when constructing a closure, the environment does not need to be restricted: it already is.

In a linear program, each mapping in the environment corresponds to the single occurrence of a bound variable. So when evaluating an application, this tightening *splits* the environment ρ into (ρ_1, ρ_2), where ρ_1 closes the operator, ρ_2 closes the operand, and $\mathbf{dom}(\rho_1) \cap \mathbf{dom}(\rho_2) = \emptyset$.

Definition 1. *Environment ρ* linearly closes *t (or $\langle t, \rho\rangle$ is a* linear closure*) iff t is linear, ρ closes t, and for all $x \in \mathbf{dom}(\rho)$, x occurs exactly once (free) in t, $\rho(x)$ is a linear closure, and for all $y \in \mathbf{dom}(\rho)$, x does not occur (free or bound) in $\rho(y)$. The* size *of a linear closure $\langle t, \rho\rangle$ is defined as:*

$$\begin{aligned}
|t, \rho| &= |t| + |\rho| \\
|x| &= 1 \\
|(\lambda x.t^{\ell})| &= 1 + |t| \\
|(t_1^{\ell_1}\ t_2^{\ell_2})| &= 1 + |t_1| + |t_2| \\
|[x_1 \mapsto c_1, \ldots, x_n \mapsto c_n]| &= n + \sum_i |c_i|
\end{aligned}$$

The following lemma states that evaluation of a linear closure cannot produce a larger value. This is the environment-based analog to the easy observation that β-reduction *strictly* decreases the size of a linear term.

Lemma 1. *If ρ linearly closes t and $\mathcal{E}[\![t^{\ell}]\!]\rho = c$, then $|c| \leq |t, \rho|$.*

Proof. Straightforward by induction on $|t, \rho|$, reasoning by case analysis on t. Observe that the size strictly decreases in the application and variable case, and remains the same in the abstraction case. □

Definition 2. *A cache $\widehat{\mathsf{C}}$* respects *$\langle t, \rho\rangle$ (written $\widehat{\mathsf{C}} \vdash t, \rho$) when,*

1. *ρ linearly closes t,*
2. *$\forall x \in \mathbf{dom}(\rho).\rho(x) = \langle t', \rho'\rangle \Rightarrow \widehat{\mathsf{C}}(x) = \{t'\}$ and $\widehat{\mathsf{C}} \vdash t', \rho'$, and*
3. *$\forall \ell \in \mathbf{lab}(t) \setminus \mathbf{fv}(t), \widehat{\mathsf{C}}(\ell) = \emptyset$.*

Clearly, $\emptyset \vdash t, \emptyset$ when t is closed and linear, i.e. t is a linear program.

Assume that the imperative algorithm $\mathcal{A}[\![\cdot]\!]$ of Section 3 is written in the obvious "cache-passing" functional style.

Theorem 1. *If $\widehat{\mathsf{C}} \vdash t, \rho$, $\widehat{\mathsf{C}}(\ell) = \emptyset$, $\ell \notin \mathbf{lab}(t, \rho)$, $\mathcal{E}[\![t^{\ell}]\!]\rho = \langle t', \rho'\rangle$, and $\mathcal{A}[\![t^{\ell}]\!]\widehat{\mathsf{C}} = \widehat{\mathsf{C}}'$, then $\widehat{\mathsf{C}}'(\ell) = \{t'\}$, $\widehat{\mathsf{C}}' \vdash t', \rho'$, and $\widehat{\mathsf{C}}' \models t^{\ell}$.*

An important consequence is noted in Corollary 1.

Proof. By induction on $|t, \rho|$, reasoning by case analysis on t.

- Case $t \equiv x$.
  Since $\widehat{\mathsf{C}} \vdash x, \rho$ and ρ linearly closes x, thus $\rho = [x \mapsto \langle t', \rho' \rangle]$ and ρ' linearly closes t'. By definition,

$$\begin{aligned}\mathcal{E}[\![x^\ell]\!]\rho &= \langle t', \rho' \rangle, \text{ and}\\ \mathcal{A}[\![x^\ell]\!]\widehat{\mathsf{C}} &= \widehat{\mathsf{C}}[x \leftrightarrow \ell].\end{aligned}$$

  Again since $\widehat{\mathsf{C}} \vdash x, \rho$, $\widehat{\mathsf{C}}(x) = \{t'\}$, with which the assumption $\widehat{\mathsf{C}}(\ell) = \emptyset$ implies

$$\widehat{\mathsf{C}}[x \leftrightarrow \ell](x) = \widehat{\mathsf{C}}[x \leftrightarrow \ell](\ell) = \{t'\},$$

  and therefore $\widehat{\mathsf{C}}[x \leftrightarrow \ell] \models x^\ell$. It remains to show that $\widehat{\mathsf{C}}[x \leftrightarrow \ell] \vdash t', \rho'$. By definition, $\widehat{\mathsf{C}} \vdash t', \rho'$. Since x and ℓ do not occur in t', ρ' by linearity and assumption, respectively, it follows that $\widehat{\mathsf{C}}[x \mapsto \ell] \vdash t', \rho'$ and the case holds.
- Case $t \equiv \lambda x.e_0$.
  By definition,

$$\begin{aligned}\mathcal{E}[\![(\lambda x.e_0)^\ell]\!]\rho &= \langle \lambda x.e_0, \rho \rangle,\\ \mathcal{A}[\![(\lambda x.e_0)^\ell]\!]\widehat{\mathsf{C}} &= \widehat{\mathsf{C}}[\ell \mapsto^+ \{\lambda x.e_0\}],\end{aligned}$$

  and by assumption $\widehat{\mathsf{C}}(\ell) = \emptyset$, so $\widehat{\mathsf{C}}[\ell \mapsto^+ \{\lambda x.e_0\}](\ell) = \{\lambda x.e_0\}$ and therefore $\widehat{\mathsf{C}}[\ell \mapsto^+ \{\lambda x.e_0\}] \models (\lambda x.e_0)^\ell$. By assumptions $\ell \notin \mathbf{lab}(\lambda x.e_0, \rho)$ and $\widehat{\mathsf{C}} \vdash \lambda x.e_0, \rho$, it follows that $\widehat{\mathsf{C}}[\ell \mapsto^+ \{\lambda x.e_0\}] \vdash \lambda x.e_0, \rho$ and the case holds.
- Case $t \equiv t_1^{\ell_1}\ t_2^{\ell_2}$. Let

$$\begin{aligned}\mathcal{E}[\![t_1]\!]\rho \upharpoonright \mathbf{fv}(t_1^{\ell_1}) &= \langle v_1, \rho_1 \rangle = \langle \lambda x.t_0^{\ell_0}, \rho_1 \rangle,\\ \mathcal{E}[\![t_2]\!]\rho \upharpoonright \mathbf{fv}(t_2^{\ell_2}) &= \langle v_2, \rho_2 \rangle,\\ \mathcal{A}[\![t_1]\!]\widehat{\mathsf{C}} &= \widehat{\mathsf{C}}_1, \text{ and}\\ \mathcal{A}[\![t_2]\!]\widehat{\mathsf{C}} &= \widehat{\mathsf{C}}_2.\end{aligned}$$

  Clearly, for $i \in \{1, 2\}$, $\widehat{\mathsf{C}} \vdash t_i, \rho \upharpoonright \mathbf{fv}(t_i)$ and

$$1 + \sum_i |t_i^{\ell_i}, \rho \upharpoonright \mathbf{fv}(t_i^{\ell_i})| = |(t_1^{\ell_1}\ t_2^{\ell_2}), \rho|.$$

  By induction, for $i \in \{1, 2\} : \widehat{\mathsf{C}}_i(\ell_i) = \{v_i\}, \widehat{\mathsf{C}}_i \vdash \langle v_i, \rho_i \rangle$, and $\widehat{\mathsf{C}}_i \models t_i^{\ell_i}$. From this, it is straightforward to observe that $\widehat{\mathsf{C}}_1 = \widehat{\mathsf{C}} \cup \widehat{\mathsf{C}}'_1$ and $\widehat{\mathsf{C}}_2 = \widehat{\mathsf{C}} \cup \widehat{\mathsf{C}}'_2$ where $\widehat{\mathsf{C}}'_1$ and $\widehat{\mathsf{C}}'_2$ are disjoint. So let $\widehat{\mathsf{C}}_3 = (\widehat{\mathsf{C}}_1 \cup \widehat{\mathsf{C}}_2)[x \leftrightarrow \ell_2]$. It is clear that $\widehat{\mathsf{C}}_3 \models t_i^{\ell_i}$. Furthermore,

$$\begin{aligned}\widehat{\mathsf{C}}_3 &\vdash t_0, \rho_1[x \mapsto \langle v_2, \rho_2 \rangle],\\ \widehat{\mathsf{C}}_3(\ell_0) &= \emptyset, \text{ and}\\ \ell_0 &\notin \mathbf{lab}(t_0, \rho_1[x \mapsto \langle v_2, \rho_2 \rangle]).\end{aligned}$$

By Lemma 1, $|v_i, \rho_i| \leq |t_i, \rho \restriction \mathbf{fv}(t_i)|$, therefore

$$|t_0, \rho_1[x \mapsto \langle v_2, \rho_2 \rangle]| < |(t_1^{\ell_1}\ t_2^{\ell_2})|.$$

Let

$$\mathcal{E}[\![t_0^{\ell_0}]\!]\rho_1[x \mapsto \langle v_2, \rho_2 \rangle] = \langle v', \rho' \rangle,$$
$$\mathcal{A}[\![t_0^{\ell_0}]\!]\widehat{\mathsf{C}}_3 = \widehat{\mathsf{C}}_4,$$

and by induction, $\widehat{\mathsf{C}}_4(\ell_0) = \{v'\}$, $\widehat{\mathsf{C}}_4 \vdash v', \rho'$, and $\widehat{\mathsf{C}}_4 \models v'$. Finally, observe that $\widehat{\mathsf{C}}_4[\ell \leftrightarrow \ell_0](\ell) = \widehat{\mathsf{C}}_4[\ell \leftrightarrow \ell_0](\ell_0) = \{v'\}$, $\widehat{\mathsf{C}}_4[\ell \leftrightarrow \ell_0] \vdash v', \rho'$, and $\widehat{\mathsf{C}}_4[\ell \leftrightarrow \ell_0] \models (t_1^{\ell_1}\ t_2^{\ell_2})^{\ell}$, so the case holds.

□

We can now establish the correspondence between analysis and evaluation.

Corollary 1. *If $\widehat{\mathsf{C}}$ is the simple closure analysis of a linear program t^{ℓ}, then $\mathcal{E}[\![t^{\ell}]\!]\emptyset = \langle v, \rho' \rangle$ where $\widehat{\mathsf{C}}(\ell) = \{v\}$ and $\widehat{\mathsf{C}} \vdash v, \rho'$.*

By a simple replaying of the proof substituting the containment constraints of 0CFA for the equality constraints of simple closure analysis, it is clear that the same correspondence can be established, and therefore 0CFA and simple closure analysis are identical for linear programs.

Corollary 2. *If e is a linear program, then $\widehat{\mathsf{C}}$ is the simple closure analysis of e iff $\widehat{\mathsf{C}}$ is the 0CFA of e.*

Discussion: Returning to our earlier question of the computationally potent ingredients in a static analysis, we can now see that when the term is linear, whether flows are directional and bidirectional is irrelevant. For these terms, simple closure analysis, 0CFA, and evaluation are equivalent. And, as we will see, when an analysis is *exact* for linear terms, the analysis will have a PTIME-hardness bound.

5 Lower Bounds for Flow Analysis

There are at least two fundamental ways to reduce the complexity of analysis. One is to compute more approximate answers, the other is to analyze a syntactically restricted language.

We use *linearity* as the key ingredient in proving lower bounds on analysis. This shows not only that simple closure analysis and other flow analyses are PTIME-complete, but the result is rather robust in the face of analysis design based on syntactic restrictions. This is because we are able to prove the lower bound via a highly restricted programming language—the linear λ-calculus. So long as the subject language of an analysis includes the linear λ-calculus, and is exact for this subset, the analysis must be at least PTIME-hard.

The decision problem answered by flow analysis, described colloquially in Section 1, is formulated as follows:

Flow Analysis Problem: Given a closed expression e, a term v, and label ℓ, is $v \in \widehat{\mathsf{C}}(\ell)$ in the analysis of e?

Theorem 2. *If analysis corresponds to evaluation on linear terms, the analysis is* PTIME*-hard.*

The proof is by reduction from the canonical PTIME-complete problem [15]:

Circuit Value Problem: Given a Boolean circuit C of n inputs and one output, and truth values $\boldsymbol{x} = x_1, \ldots, x_n$, is $\boldsymbol{x}$ accepted by C?

An instance of the circuit value problem can be compiled, using only logarithmic space, into an instance of the flow analysis problem following the construction in [9]. Briefly, the circuit and its inputs are compiled into a linear λ-term, which simulates C on $\boldsymbol{x}$ via *evaluation*—it normalizes to true if C accepts $\boldsymbol{x}$ and false otherwise. But since the analysis faithfully captures evaluation of linear terms, and our encoding is linear, the circuit can be simulated by flow analysis.

The encodings work like this: *tt* is the identity on pairs, and *ff* is the swap. Boolean values are either $\langle tt, ff\rangle$ or $\langle ff, tt\rangle$, where the first component is the "real" value, and the second component is the complement.

$$\begin{array}{ll} tt \equiv \lambda p.\text{let } \langle x, y\rangle = p \text{ in } \langle x, y\rangle & \quad \mathit{True} \equiv \langle tt, ff\rangle \\ ff \equiv \lambda p.\text{let } \langle x, y\rangle = p \text{ in } \langle y, x\rangle & \quad \mathit{False} \equiv \langle ff, tt\rangle \end{array}$$

The simplest connective is *Not*, which is an inversion on pairs, like *ff*. A *linear* copy connective is defined as:

$$\mathit{Copy} \equiv \lambda b.\text{let } \langle u, v\rangle = b \text{ in } \langle u\langle tt, ff\rangle, v\langle ff, tt\rangle\rangle.$$

The coding is easily explained: suppose b is *True*, then u is identity and v twists; so we get the pair $\langle \mathit{True}, \mathit{True}\rangle$. Suppose b is *False*, then u twists and v is identity; we get $\langle \mathit{False}, \mathit{False}\rangle$.

The *And* connective is defined as:

$$\begin{array}{l} \mathit{And} \equiv \lambda b_1.\lambda b_2. \\ \qquad \text{let } \langle u_1, v_1\rangle = b_1 \text{ in} \\ \qquad \text{let } \langle u_2, v_2\rangle = b_2 \text{ in} \\ \qquad \text{let } \langle p_1, p_2\rangle = u_1\langle u_2, ff\rangle \text{ in} \\ \qquad \text{let } \langle q_1, q_2\rangle = v_1\langle tt, v_2\rangle \text{ in} \\ \qquad\quad \langle p_1, q_1 \circ p_2 \circ q_2 \circ ff\rangle. \end{array}$$

Conjunction works by computing pairs $\langle p_1, p_2\rangle$ and $\langle q_1, q_2\rangle$. The former is the usual conjuction on the first components of the Booleans b_1, b_2: $u_1\langle u_2, ff\rangle$ can be read as "if u_1 then u_2, otherwise false (*ff*)." The latter is (exploiting deMorgan duality) the disjunction of the complement components of the Booleans: $v_1\langle tt, v_2\rangle$ is read as "if v_1 (i.e. if not u_1) then true (*tt*), otherwise v_2 (i.e. not u_2)." The result of the computation is equal to $\langle p_1, q_1\rangle$, but this leaves p_2, q_2 unused, which would violate linearity. However, there is symmetry to this *garbage*, which allows for its disposal. Notice that, while we do not know whether p_2 is *tt* or *ff* and similarly for q_2, we do know that *one of them is tt while the other is ff*. Composing the two together, we are guaranteed that $p_2 \circ q_2 = \mathit{ff}$. Composing this again with another twist (ff) results in the identity function $p_2 \circ q_2 \circ \mathit{ff} = \mathit{tt}$. Finally, composing this with q_1 is just equal to q_1, so $\langle p_1, q_1 \circ p_2 \circ q_2 \circ \mathit{ff} \rangle = \langle p_1, q_1 \rangle$, which is the desired result, but the symmetric garbage has been *annihilated*, maintaining linearity.

This hacking, with its self-annihilating garbage, is an improvement over that given in [16] and allows Boolean computation without K-redexes, making the lower bound stronger, but also preserving all flows. In addition, it is the best way to do circuit computation in multiplicative linear logic, and is how you compute similarly in non-affine typed λ-calculus.

We know from Corollary 1 that normalization and analysis of linear programs are synonymous, and our encoding of circuits will faithfully simulate a given circuit on its inputs, evaluating to true iff the circuit accepts its inputs. But it does not immediately follow that the circuit value problem can be reduced to the flow analysis problem. Let $||C, \boldsymbol{x}||$ be the encoding of the circuit and its inputs. It is tempting to think the instance of the flow analysis problem could be stated:

$$\text{is } \mathit{True} \text{ in } \widehat{\mathsf{C}}(\ell) \text{ in the analysis of } ||C, \boldsymbol{x}||^{\ell}?$$

The problem with this is there may be many syntactic instances of "*True*." Since the flow analysis problem must ask about a particular one, this reduction will not work. The fix is to use a context which expects a boolean expression and induces a particular flow (that can be asked about in the flow analysis problem) iff that expression evaluates to a true value [9].

Corollary 3. *Simple closure analysis is* PTIME*-complete.*

6 Other Monovariant Analyses

In this section, we survey some of the existing monovariant analyses that either approximate or restrict 0CFA to obtain faster analysis times. In each case, we sketch why these analyses are complete for PTIME.

6.1 Ashley and Dybvig's Sub-0CFA

In [8], Ashley and Dybvig develop a general framework for specifying and computing flow analyses, which can be instantiated to obtain 0CFA or Jagannathan and Weeks' polynomial 1CFA [17], for example. They also develop a class of instantiations of their framework dubbed *sub-0CFA* that is faster to compute, but less accurate than 0CFA.

This analysis works by explicitly bounding the number of times the cache can be updated for any given program point. After this threshold has been crossed, the cache is updated with a distinguished *unknown* value that represents all possible λ-abstractions in the program. Bounding the number of updates to the cache for any given location effectively bounds the number of passes over the program an analyzer must make, producing an analysis that is $O(n)$ in the size of the program. Empirically, Ashley and Dybvig observe that setting the bound to 1 yields an inexpensive analysis with no significant difference in enabling optimizations with respect to 0CFA.

The idea is the cache gets updated once (n times in general) before giving up and saying all λ-abstractions flow out of this point. But for a linear term, the cache is only updated at most once for each program point. Thus we conclude even when the sub-0CFA bound is 1, the problem is PTIME-complete.

As Ashley and Dybvig note, for any given program, there exists an analysis in the sub-0CFA class that is identical to 0CFA (namely by setting n to the number of passes 0CFA makes over the given program). We can further clarify this relationship by noting that for all linear programs, all analyses in the sub-0CFA class are identical to 0CFA (and thus simple closure analysis).

6.2 Subtransitive 0CFA

Heintze and McAllester [6] have shown that the "cubic bottleneck" of computing full 0CFA—that is, computing all the flows in a program—cannot be avoided in general without combinatorial breakthroughs: the problem is 2NPDA-hard, for which the "the cubic time decision procedure [...] has not been improved since its discovery in 1968."

Given the unlikeliness of improving the situation in general, Heintze and McAllester [10] identify several simpler flow questions (including the decision problem discussed in the paper, which is the simplest; answers to any of the other questions imply an answer to this problem). They give algorithms for simply typed terms that answer these restricted flow problems, which under certain conditions, compute in less than cubic time.

Their analysis is linear with respect to a program's graph, which in turn, is bounded by the size of the program's type. Thus, bounding the size of a program's type results in a linear bound on the running times of these algorithms. If this type bound is removed, though, it is clear that even these simplified flow problems (and their bidirectional-flow analogs), are complete for PTIME: observe that every linear term is simply typable, however in our lower bound construction, the type size is proportional to the size of the circuit being simulated. As they point out, when type size is not bounded, the flow graph may be exponentially larger than the program, in which case the standard cubic algorithm is preferred.

Independently, Mossin [18] developed a type-based analysis that, under the assumption of a constant bound on the size of a program's type, can answer restricted flow questions such as single source/use in linear time with respect to the size of the explicitly typed program. But again, removing this imposed bound results in PTIME-completeness.

As Hankin *et al.* [19] point out: both Heintze and McAllester's and Mossin's algorithms operate on type structure (or structure isomorphic to type structure), but with either implicit or explicit η-expansion. For simply typed terms, this can result in an exponential blow-up in type size. It is not surprising then, that given a much richer graph structure, the analysis can be computed quickly. In this light, recent results on 0CFA of η-expanded, simply typed programs can be seen as an improvement of the subtransitive flow analysis since it works equally well for languages with first-class control and can be performed with only a fixed number of pointers into the program structure, i.e. it is computable in LOGSPACE (and in other words, PTIME = LOGSPACE up to η) [9].

7 Conclusions and Perspective

When an analysis is *exact*, it will be possible to establish a correspondence with evaluation. The richer the language for which analysis is exact, the harder it will be to compute the analysis. As an example in the extreme, Mossin [20] developed a flow analysis that is exact for simply typed terms. The computational resources that may be expended to compute this analysis are *ipso facto* not bounded by any elementary recursive function [21]. However, most flow analyses do not approach this kind of expressivity. By way of comparison, 0CFA only captures PTIME, and yet researchers have still expending a great deal of effort deriving approximations to 0CFA that are faster to compute. But as we have shown for a number of them, they all coincide on linear terms, and so they too capture PTIME.

We should be clear about what is being said, and not said. There is a considerable difference in practice between linear algorithms (nominally considered efficient) and cubic algorithms (still feasible, but taxing for large inputs), even though both are polynomial-time. PTIME-completeness does not distinguish the two. But if a sub-polynomial (e.g., LOGSPACE) algorithm was found for this sort of flow analysis, it would depend on (or lead to) things we do not know (LOGSPACE = PTIME). Likewise, were a parallel implementation of this flow analysis to run in logarithmic time (i.e., NC), we would consequently be able to parallelize every polynomial time algorithm similarly.

A fundamental question we need to be able to answer is this: what can be deduced about a long-running program with a time-bounded analyzer? When we statically analyze exponential-time programs with a polynomial-time method, there should be a analytic bound on what we can learn at compile-time: a theorem delineating how exponential time is being viewed through the compressed, myopic lens of polynomial time computation.

For example, a theorem due to Statman [21] says this: let **P** be a property of simply-typed λ-terms that we would like to detect by static analysis, where **P** is invariant under reduction (normalization), and is computable in elementary time (polynomial, or exponential, or doubly-exponential, or...). Then **P** is a *trivial* property: for any type τ, **P** is satisfied by *all* or *none* of the programs of type τ. Henglein and Mairson [22] have complemented these results, showing that if a property is invariant under β-reduction for a class of programs that can encode all Turing Machines solving problems of complexity class F using reductions from complexity class G, then any superset is either F-complete or trivial. Simple typability has this property for linear and linear affine λ-terms [16,22], and these terms are sufficient to code all polynomial-time Turing Machines.

We would like to prove some analogs of these theorems, with or without the typing condition, but weakening the condition of "invariant under reduction" to some *approximation* analogous to the approximations of flow analysis, as described above. We are motivated as well by yardsticks such as Shannon's theorem from information theory [23]: specify a bandwidth for communication and an error rate, and Shannon's results give bounds on the channel capacity. We too have essential measures: the time complexity of our analysis, the asymptotic differential between that bound and the time bound of the program we are analyzing. There ought to be a fundamental result about what information can be yielded as a function of that differential. At one end, if the program and analyzer take the same time, the analyzer can just run the program to find out everything. At the other end, if the analyzer does no work (or a constant amount of work), nothing can be learned. Analytically speaking, what is in between?

Acknowledgments. We are grateful to Olin Shivers and Matt Might for a long, fruitful, and ongoing dialogue on flow analysis. We thank the anonymous reviewers for insightful comments. The first author also thanks the researchers of the Northeastern University Programming Research Lab for the hospitality and engaging discussions had as a visiting lecturer over the last year.

References

1. Jones, N.D.: Flow analysis of lambda expressions (preliminary version). In: Proceedings of the 8th Colloquium on Automata, Languages and Programming, London, UK, pp. 114–128. Springer, Heidelberg (1981)
2. Sestoft, P.: Replacing function parameters by global variables. Master's thesis, DIKU, University of Copenhagen, Denmark, Master's thesis no. 254 (1988)
3. Shivers, O.: Control-Flow Analysis of Higher-Order Languages, or Taming Lambda. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Report CMU-CS-91-145 (1991)
4. Midtgaard, J.: Control-flow analysis of functional programs. Technical Report BRICS RS-07-18, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark (2007)
5. Shivers, O.: Control flow analysis in Scheme. In: PLDI 1988: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, pp. 164–174. ACM, New York (1988)
6. Heintze, N., McAllester, D.: On the cubic bottleneck in subtyping and flow analysis. In: LICS 1997: Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science, Washington, DC, USA, p. 342. IEEE Computer Society, Los Alamitos (1997)
7. Henglein, F.: Simple closure analysis. DIKU Semantics Report D-193 (1992)
8. Ashley, J.M., Dybvig, R.K.: A practical and flexible flow analysis for higher-order languages. ACM Trans. Program. Lang. Syst. 20(4), 845–868 (1998)
9. Van Horn, D., Mairson, H.G.: Relating complexity and precision in control flow analysis. In: Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming, pp. 85–96. ACM Press, New York (2007)

10. Heintze, N., McAllester, D.: Linear-time subtransitive control flow analysis. In: PLDI 1997: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation, pp. 261–272. ACM, New York (1997)
11. Girard, J.Y.: Linear logic: its syntax and semantics. In: Proceedings of the workshop on Advances in linear logic. Cambridge University Press, Cambridge (1995)
12. Sestoft, P.: Replacing function parameters by global variables. In: FPCA 1989: Proceedings of the fourth international conference on Functional programming languages and computer architecture, pp. 39–53. ACM, New York (1989)
13. Mossin, C.: Flow Analysis of Typed Higher-Order Programs. PhD thesis, DIKU, University of Copenhagen (1997)
14. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, New York (1999)
15. Ladner, R.E.: The circuit value problem is log space complete for P. SIGACT News 7(1), 18–20 (1975)
16. Mairson, H.G.: Linear lambda calculus and PTIME-completeness. Journal of Functional Programming 14(6), 623–633 (2004)
17. Jagannathan, S., Weeks, S.: A unified treatment of flow analysis in higher-order languages. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 393–407. ACM Press, New York (1995)
18. Mossin, C.: Higher-order value flow graphs. Nordic J. of Computing 5(3), 214–234 (1998)
19. Hankin, C., Nagarajan, R., Sampath, P.: Flow analysis: games and nets. In: The essence of computation: complexity, analysis, transformation, pp. 135–156. Springer, New York (2002)
20. Mossin, C.: Exact flow analysis. In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, pp. 250–264. Springer, Heidelberg (1997)
21. Statman, R.: The typed λ-calculus is not elementary recursive. Theor. Comput. Sci. 9, 73–81 (1979)
22. Henglein, F., Mairson, H.G.: The complexity of type inference for higher-order lambda calculi. In: POPL 1991: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 119–130. ACM, New York (1991)
23. Shannon, C.E.: A mathematical theory of communication. Bell System Technical Journal 27 (1948)

Quantum Entanglement Analysis Based on Abstract Interpretation

Simon Perdrix

Oxford University Computing Laboratory
simon.perdrix@comlab.ox.ac.uk

Abstract. Entanglement is a non local property of quantum states which has no classical counterpart and plays a decisive role in quantum information theory. Several protocols, like the teleportation, are based on quantum entangled states. Moreover, any quantum algorithm which does not create entanglement can be efficiently simulated on a classical computer. The exact role of the entanglement is nevertheless not well understood. Since an exact analysis of entanglement evolution induces an exponential slowdown, we consider approximative analysis based on the framework of abstract interpretation. In this paper, a concrete quantum semantics based on superoperators is associated with a simple quantum programming language. The representation of entanglement, i.e. the design of the abstract domain is a key issue. A representation of entanglement as a partition of the memory is chosen. An abstract semantics is introduced, and the soundness of the approximation is proven.

1 Introduction

Quantum entanglement is a non local property of quantum mechanics. The entanglement reflects the ability of a quantum system composed of several subsystems, to be in a state which cannot be decomposed into the states of the subsystems. Entanglement is one of the properties of quantum mechanics which caused Einstein and others to dislike the theory. In 1935, Einstein, Podolsky, and Rosen formulated the EPR paradox [7].

On the other hand, quantum mechanics has been highly successful in producing correct experimental predictions, and the strong correlations associated with the phenomenon of quantum entanglement have been observed indeed [2].

Entanglement leads to correlations between subsystems that can be exploited in information theory (e.g., teleportation scheme [3]). The entanglement plays also a decisive, but not yet well-understood, role in quantum computation, since any quantum algorithm can be efficiently simulated on a classical computer when the quantum memory is not entangled during all the computation. As a consequence, interesting quantum algorithms, like Shor's algorithm for factorisation [19], exploit this phenomenon.

In order to know what is the amount of entanglement of a quantum state, several measures of entanglement have been introduced (see for instance [13]). Recent works consist in characterising, in the framework of the one-way quantum

© Springer-Verlag Berlin Heidelberg 2008

computation [20], the amount of entanglement necessary for a universal model of quantum computation. Notice that all these techniques consist in analysing the entanglement of a given state, starting with its mathematical description.

In this paper, the entanglement *evolution* during the computation is analysed. The description of quantum evolutions is done via a simple quantum programming language. The development of such quantum programming languages is recent, see [17,8] for a survey on this topic.

An exact analysis of entanglement evolution induces an exponential slowdown of the computation. Model checking techniques have been introduced [9] including entanglement. Exponential slowdown of such analysis is avoided by reducing the domain to stabiliser states (i.e. a subset of quantum states that can be efficiently simulated on a classical computer). As a consequence, any quantum program that cannot be efficiently simulated on a classical computer cannot be analysed.

Prost and Zerrari [16] have recently introduced a logical entanglement analysis for functional languages. This logical framework allows analysis of higher-order functions, but does not provide any static analysis for the quantum programs without annotation. Moreover, only pure quantum states are considered.

In this paper, we introduce a novel approach of entanglement analysis based on the framework of abstract interpretation [5]. A concrete quantum semantics based on superoperators is associated with a simple quantum programming language. The representation of entanglement, i.e. the design of the abstract domain is a key issue. A representation of entanglement as a partition of the memory is chosen. An abstract semantics is introduced, and the soundness of the approximation is proved.

2 Basic Notions and Entanglement

2.1 Quantum Computing

We briefly recall the basic definitions of quantum computing; please refer to Nielsen and Chuang [13] for a complete introduction to the subject.

The state of a quantum system can be described by a density matrix, i.e. a self adjoint[1] positive-semidefinite[2] complex matrix of trace[3] less than one. The set of density matrices of dimension n is $D_n \subseteq \mathbb{C}^{n\times n}$.

The basic unit of information in quantum computation is a quantum bit or *qubit*. The state of a single qubit is described by a 2×2 density matrix $\rho \in D_2$. The state of a register composed of n qubits is a $2^n \times 2^n$ density matrix. If two registers A and B are in states $\rho_A \in D_{2^n}$ and $\rho_B \in D_{2^m}$, the composed system A, B is in state $\rho_A \otimes \rho_B \in D_{2^{n+m}}$.

The basic operations on quantum states are unitary operations and measurements. A unitary operation maps an n-qubit state to an n-qubit state, and is

[1] M is self adjoint (or Hermitian) if and only if $M^\dagger = M$.

[2] M is positive-semidefinite if all the eigenvalues of M are non-negative.

[3] The trace of M ($\mathrm{tr}(M)$) is the sum of the diagonal elements of M.

given by a $2^n \times 2^n$-unitary matrix[4]. If a system in state ρ evolves according to a unitary transformation U, the resulting density matrix is $U\rho U^\dagger$. The parallel composition of two unitary transformations U_A, U_B is $U_A \otimes U_B$.

The following unitary transformations form an approximative universal family of unitary transformations, i.e. any unitary transformation can be approximated by composing the unitary transformations of the family [13].

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}, CNot = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

A measurement is described by a family of projectors $\{\mathrm{P}_x, x \in X\}$ satisfying $\mathrm{P}_i^2 = \mathrm{P}_i$, $\mathrm{P}_i\mathrm{P}_j = 0$ if $i \neq j$, and $\sum_{x\in X} \mathrm{P}_x = \mathbb{I}$. A computational basis measurement is $\{\mathrm{P}_k, 0 \leq k < 2^n\}$, where P_k has 0 entries everywhere except one 1 at row k, column k. The parallel composition of two measurements $\{\mathrm{P}_x, x \in X\}$, $\{\mathrm{P}'_y, y \in Y\}$ is $\{\mathrm{P}_x \otimes \mathrm{P}'_y, (x, y) \in X \times Y\}$.

According to a probabilistic interpretation, a measurement according to $\{\mathrm{P}_x, x \in X\}$ of a state ρ produces the classical outcome $x \in X$ with probability $\mathrm{tr}(\mathrm{P}_x\rho\mathrm{P}_x)$ and transforms ρ into $\frac{1}{\mathrm{tr}(\mathrm{P}_x\rho\mathrm{P}_x)}\mathrm{P}_x\rho\mathrm{P}_x$.

Density matrices is a useful formalism for representing probability distributions of quantum states, since the state ρ of a system which is in state ρ_1 (resp. ρ_2) with probability p_1 (resp. p_2) is $\rho = p_1\rho_1 + p_2\rho_2$. As a consequence, a measurement according to $\{\mathrm{P}_x, x \in X\}$ transforms ρ into $\sum_{x\in X} \mathrm{P}_x\rho\mathrm{P}_x$.

Notice that the sequential compositions of two measurements (or of a measurement and a unitary transformation) is no more a measurement nor a unitary transformation, but a superoperator, i.e. a trace-decreasing[5] completely positive[6] linear map. Any quantum evolution can be described by a superoperator.

The ability to initialise any qubit in a given state ρ_0, to apply any unitary transformation from a universal family, and to perform a computational measurement are enough for simulating any superoperator.

2.2 Entanglement

Quantum entanglement is a non local property which has no classical counterpart. Intuitively, a quantum state of a system composed of several subsystems is

[4] U is unitary if and only if $U^\dagger U = UU^\dagger = \mathbb{I}$.

[5] F is trace decreasing iff $\mathrm{tr}(F(\rho)) \leq \mathrm{tr}(\rho)$ for any ρ in the domain of F. Notice that superoperators are sometimes defined as trace-perserving maps, however trace-decreasing is more suitable in a semantical context, see [18] for details.

[6] F is positive if $F(\rho)$ is positive-semidefinite for any positive ρ in the domain of F. F is completely positive if $\mathbb{I}_k \otimes F$ is positive for any k, where $\mathbb{I}_k : \mathbb{C}^{k\times k} \to \mathbb{C}^{k\times k}$ is the identity map.

entangled if it cannot be decomposed into the state of its subsystems. A quantum state which is not entangled is called *separable*.

More precisely, for a given finite set of qubits Q, let $n = |Q|$. For a given partition A, B of Q, and a given $\rho \in D_{2^n}$, ρ is biseparable according to A, B (or (A, B)-separable for short) if and only if there exist K, $p_k \geq 0$, ρ_k^A and ρ_k^B such that

$$\rho = \sum_{k \in K} p_k \rho_k^A \otimes \rho_k^B$$

ρ is entangled according to the partition A, B if and only if ρ is not (A, B)-separable.

Notice that biseparability provides a very partial information about the entanglement of a quantum state, for instance for a 3-qubit state ρ, which is $(\{1\}, \{2, 3\})$-separable, qubit 2 and qubit 3 may be entangled or not.

One way to generalise the biseparability is to consider that a quantum state is π-separable – where $\pi = \{Q_j, j \in J\}$ is a partition of Q – if and only if there exist K, $p_k \geq 0$, and $\rho_k^{Q_j}$ such that

$$\rho = \sum_{k \in K} p_k \left(\bigotimes_{j \in J} \rho_k^{Q_j} \right)$$

Notice that the structure of quantum entanglement presents some interesting and non trivial properties. For instance there exist some 3-qubit states ρ such that ρ is bi-separable for any bi-partition of the 3 qubits, but not fully separable i.e., separable according to the partition $\{\{1\}, \{2\}, \{3\}\}$. As a consequence, for a given quantum state, there is not necessary a *best representation* of its entanglement.

2.3 Standard and Diagonal Basis

For a given state $\rho \in \mathcal{D}^Q$ and a given qubit $q \in Q$, if ρ is $(\{q\}, Q \setminus \{q\})$-separable, then q is separated from the rest of the memory. Moreover, such a qubit may be a basis state in the standard basis ($\mathbf{s}$) or the diagonal basis ($\mathbf{d}$), meaning that the state of this qubit can be seen as a 'classical state' according to the corresponding basis.

More formally, a qubit q of ρ is in the standard basis if there exists $p_0, p_1 \geq 0$, and $\rho_0, \rho_1 \in \mathcal{D}^{Q \setminus \{q\}}$ such that $\rho = p_0 \mathrm{P}_0 \otimes \rho_0 + p_1 \mathrm{P}_1 \otimes \rho_1$. Equivalently, q is in the standard basis if and only if $\mathrm{P}_0^{(q)} \rho \mathrm{P}_1^{(q)} = \mathrm{P}_1^{(q)} \rho \mathrm{P}_0^{(q)} = 0$, where $\mathrm{P}_k^{(q)} = \mathrm{P}_k^{\{q\}} \otimes \mathbb{I}^{Q \setminus \{q\}}$ meaning that P_k is applied on qubit q. A qubit q is in the diagonal basis in ρ if and only if q is in the standard basis in $H^{(q)} \rho H^{(q)}$, where $H^{(q)} = H^{\{q\}} \otimes \mathbb{I}^{Q \setminus \{q\}}$.

Notice that some states, like the maximally mixed 1-qubit state $\frac{1}{2}(\mathrm{P}_0 + \mathrm{P}_1)$ are in both standard and diagonal basis, while others are neither in standard nor diagonal basis like the 1-qubit state $TH\mathrm{P}_0HT$.

We introduce a function $\beta : \mathcal{D}^Q \to B^Q$, where $B^Q = Q \to \{\mathbf{s}, \mathbf{d}, \top, \bot\}$, such that $\beta(\rho)$ describes which qubits of ρ are in the standard or diagonal basis:

Definition 1. *For any finite Q, let $\beta : \mathcal{D}^Q \to B^Q$ such that for any $\rho \in \mathcal{D}^Q$, and any $q \in Q$,*

$$\beta(\rho)_q = \begin{cases} \bot & \text{if } q \text{ is in both standard and diagonal basis in } \rho \\ \mathbf{s} & \text{if } q \text{ is in the standard and not in the diagonal basis in } \rho \\ \mathbf{d} & \text{if } q \text{ is in the diagonal and not in the standard basis in } \rho \\ \top & \text{otherwise} \end{cases}$$

3 A Quantum Programming Language

Several quantum programming languages have been introduced recently. For a complete overview see [8]. We use an imperative quantum programming language introduced in [15], the syntax is similar to the language introduced by Abramsky [1]. For the sake of simplicity and in order to focus on entanglement analysis, the memory is supposed to be fixed and finite. Moreover, the memory is supposed to be composed of qubits only, whereas hybrid memories composed of classical and quantum parts are often considered. However, contrary to the quantum circuit or quantum Turing machine frameworks, the absence of classical memory does not avoid the classical control of the quantum computation since classically-controlled conditional structures are allowed (see section 3.1.)

Definition 2 (Syntax). *For a given finite set of symbols $q \in Q$, a program is a pair $\langle C, Q\rangle$ where C is a command defined as follows:*

$$\begin{aligned} C ::= \ & \mathsf{skip} \\ & \mid C_1; C_2 \\ & \mid \mathsf{if}\ q\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \\ & \mid \mathsf{while}\ q\ \mathsf{do}\ C \\ & \mid \mathsf{H}(q) \\ & \mid \mathsf{T}(q) \\ & \mid \mathsf{CNot}(q, q) \end{aligned}$$

Example 1. Quantum entanglement between two qubits q_2 and q_3 can be created for instance by applying H and $CNot$ on an appropriate state. Such an entangled state can then be used to *teleporte* the state of a third qubit q_1. The protocol of teleportation [3] can be described as $\langle \mathsf{teleportation}, \{q_1, q_2, q_3\}\rangle$, where

```
teleportation : H($q_2$);
                CNot($q_2, q_3$);
                CNot($q_1, q_2$);
                H($q_1$);
                if $q_1$ then
                   if $q_2$ then skip else $\sigma_x(q_3)$
                else
                   if $q_2$ then $\sigma_z(q_3)$ else $\sigma_y(q_3)$
```

The semantics of this program is given in example 2.

3.1 Concrete Semantics

Several domains for quantum computation have been introduced [1,12,14]. Among them, the domain of superoperators over density matrices, introduced by Selinger [18] turns out to be one of the most adapted to quantum semantics. Thus, we introduce a denotational semantics following the work of Selinger.

For a finite set of variables $Q = \{q_0, \ldots, q_n\}$, let $\mathcal{D}^Q = D_{2^{|Q|}}$. Q is a set of qubits, the state of Q is a density operator in $\mathcal{D}^Q$.

Definition 3 (Löwner partial order). *For matrices M and N in $\mathbb{C}^{n\times n}$, $M \sqsubseteq N$ if $N - M$ is positive-semidefinite.*

In [18], Selinger proved that the poset $(\mathcal{D}^Q, \sqsubseteq)$ is a complete partial order with 0 as its least element. Moreover the poset of superoperators over $\mathcal{D}^Q$ is a complete partial order as well, with 0 as least element and where the partial order $\sqsubseteq'$ is defined as $F \sqsubseteq' G \iff \forall k \geq 0, \forall \rho \in \mathcal{D}_{k2^{|Q|}}, (\mathbb{I}_k \otimes F)(\rho) \sqsubseteq (\mathbb{I}_k \otimes G)(\rho)$, where $\mathbb{I}_k : \mathcal{D}_k \to \mathcal{D}_k$ is the identity map. Notice that these complete partial orders are not lattices (see [18].)

We are now ready to introduce the concrete denotational semantics which associates with any program $\langle C, Q\rangle$, a superoperator $[\![C]\!] : \mathcal{D}^Q \to \mathcal{D}^Q$.

Definition 4 (Denotational semantics).

$$[\![\mathsf{skip}]\!] = \mathbb{I}$$

$$[\![C_1; C_2]\!] = [\![C_2]\!] \circ [\![C_1]\!]$$

$$[\![\mathsf{U}(q)]\!] = \lambda\rho. U_q \rho U_q^\dagger$$

$$[\![\mathsf{CNot}(q_1, q_2)]\!] = \lambda\rho. CNot_{q_1,q_2} \rho CNot_{q_1,q_2}^\dagger$$

$$[\![\mathsf{if}\ q\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ ]\!] = \lambda\rho. \left([\![C_1]\!](\mathrm{P}_q^{\mathsf{true}} \rho \mathrm{P}_q^{\mathsf{true}}) + [\![C_2]\!](\mathrm{P}_q^{\mathsf{false}} \rho \mathrm{P}_q^{\mathsf{false}})\right)$$

$$\begin{aligned}[\![\mathsf{while}\ q\ \mathsf{do}\ C\ ]\!] &= \mathsf{lfp}\left(\lambda f.\lambda\rho. \left(f \circ [\![C]\!](\mathrm{P}_q^{\mathsf{true}} \rho \mathrm{P}_q^{\mathsf{true}}) + \mathrm{P}_q^{\mathsf{false}} \rho \mathrm{P}_q^{\mathsf{false}}\right)\right) \\ &= \textstyle\sum_{n\in\mathbb{N}} (F_{\mathrm{P}^{\mathsf{false}}} \circ ([\![C]\!] \circ F_{\mathrm{P}^{\mathsf{true}}})^n)\end{aligned}$$

where $\mathrm{P}^{\mathsf{false}} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$ *and* $\mathrm{P}^{\mathsf{true}} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$, $F_M = \lambda\rho. M\rho M^\dagger$, *and* $M_q = M^{\{q\}} \otimes \mathbb{I}^{Q\setminus\{q\}}$ *(and* $CNot_{q_1,q_2} = CNot^{\{q_1,q_2\}} \otimes \mathbb{I}^{Q\setminus\{q_1,q_2\}}$*) meaning that M is applied on qubit q. We refer the reader to an extended version of this paper for the technical explanations on continuity and convergence.*

In the absence of classical memory, the classical control is encoded into the conditional structure $\mathsf{if}\ q\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ such that the qubit q is first measured according to the computational basis. If the first projector is applied, then the classical outcome is interpreted as *true* and the command C_1 is applied. Otherwise, the second projector is applied, and the command C_2 is performed.

According to the encoding of probabilistic evolutions in the formalism of density matrices, the overall evolution is the sum of the possible evolutions (see section 2.1.) The classical control appears in the loop $\mathsf{while}\ q\ \mathsf{do}\ C$ as well.

As a consequence of the classical control, non unitary transformations can be implemented:

$$[\![\mathsf{if}\ q\ \mathsf{then}\ q\ \mathsf{else}\ \sigma_x(q)\ ]\!] : \mathcal{D}^{\{q\}} \to \mathcal{D}^{\{q\}} = \lambda\rho.\mathrm{P}^{\mathsf{true}}$$

$$[\![\mathsf{while}\ q\ \mathsf{do}\ \mathsf{H}(q)\ ]\!] : \mathcal{D}^{\{q\}} \to \mathcal{D}^{\{q\}} = \lambda\rho.\mathrm{P}^{\mathsf{false}}$$

Notice that the matrices $\mathrm{P}^{\mathsf{true}}$ and $\mathrm{P}^{\mathsf{false}}$, used in definition 4 for describing the computational measurement $\{\mathrm{P}^{\mathsf{true}}, \mathrm{P}^{\mathsf{false}}\}$ can also be used as density matrices for describing a quantum state as above.

Moreover, notice that all the ingredients for approximating any superoperators can be encoded into the language: the ability to initialise any qubit in a given state (for instance $\mathrm{P}^{\mathsf{true}}$ or $\mathrm{P}^{\mathsf{false}}$); an approximative universal family of unitary transformation $\{H, T, CNot, \sigma_x, \sigma_y, \sigma_z\}$; and the computational measurement of a qubit q with $\mathsf{if}\ q\ \mathsf{then\ skip\ else\ skip}$.

Example 2. The program $\langle \mathsf{teleportation}, \{q_1, q_2, q_3\}\rangle$ described in example 1 realises the teleportation from q_1 to q_3, when the qubits q_2 and q_3 are both initialised in state $\mathrm{P}^{\mathsf{true}}$: for any $\rho \in \mathcal{D}_2$,

$$[\![\mathsf{teleportation}]\!](\rho \otimes \mathrm{P}^{\mathsf{true}} \otimes \mathrm{P}^{\mathsf{true}}) = \left(\frac{1}{4} \sum_{k,l \in \{\mathsf{true},\mathsf{false}\}} \mathrm{P}^k \otimes \mathrm{P}^l\right) \otimes \rho$$

4 Entanglement Analysis

What is the role of the entanglement in quantum information theory? How does the entanglement evolve during a quantum computation? We consider the problem of analysing the entanglement evolution on a classical computer, since no large scale quantum computer is available at the moment. Entanglement analysis using a quantum computer is left to further investigations[7].

In the absence of quantum computer, an obvious solution consists in simulating the quantum computation on a classical computer. Unfortunately, the classical memory required for the simulation is exponentially large in the size of the quantum memory of the program simulated. Moreover, the problem **SEP** of deciding whether a given quantum state ρ is biseparable or not is NP Hard[8] [10].

[7] Notice that this is not clear that the use of a quantum computer avoids the use of the classical computer since there is no way to measure the entanglement of a quantum state without transforming the state.

[8] For pure quantum states (i.e. $\mathrm{tr}(\rho^2) = \mathrm{tr}(\rho)$), a linear algorithm have been introduced [11] to solve the sub-problem of finding biseparability of the form $(\{q_0, \dots, q_k\}, \{q_{k+1}, \dots, q_n\})$ – thus sensitive to the ordering of the qubits in the register. Notice that this algorithm is linear in the size of the input which is a density matrix, thus the algorithm is exponential in the number of qubits.

Furthermore, the input of the problem **SEP** is a density matrix, which size is exponential in the number of qubits. As a consequence, the solution of a classical simulation is not suitable for an efficient entanglement analysis.

To tackle this problem, a solution consists in reducing the size of the quantum state space by considering a subspace of possible states, such that there exist algorithms to decide whether a state of the subspace is entangled or not in a polynomial time in the number of qubits. This solution has been developed in [9], by considering stabiliser states only. However, this solution, which may be suitable for some quantum protocols, is questionable for analysing quantum algorithms since all the quantum programs on which such an entanglement analysis can be driven are also efficiently simulable on a classical computer.

In this paper, we introduce a novel approach which consists in approximating the entanglement evolution of the quantum memory. This solution is based on the framework of abstract interpretation introduced by Cousot and Cousot [5]. Since a classical domain for driving a sound and complete analysis of entanglement is exponentially large in the number n of qubits, we consider an abstract domain of size n and we introduce an abstract semantics which leads to a sound approximation of the entanglement evolution during the computation.

4.1 Abstract Semantics

The entanglement of a quantum state can be represented as a partition of the qubits of the state (see section 2.2), thus a natural abstract domain is a domain composed of partitions. Moreover, for a given state ρ, one can add a flag for each qubit q, indicating whether the state of this qubit is in the standard basis $\mathbf{s}$ or in the diagonal basis $\mathbf{d}$ (see section 2.3).

Definition 5 (Abstract Domain). *For a finite set of variables Q, let $\mathcal{A}^Q = B^Q \times \Pi^Q$ be an abstract domain, where $B^Q = Q \to \{\mathbf{s}, \mathbf{d}, \top, \bot\}$ and Π^Q is the set of partitions of Q:*

$$\Pi^Q = \{\pi \subseteq \wp(Q) \setminus \{\emptyset\} \mid \bigcup_{X \in \pi} X = Q \text{ and } (\forall X, Y \in \pi,\ X \cap Y = \emptyset \text{ or } X = Y)\}$$

The abstract domain $\mathcal{A}$ is ordered as follows. First, let $(\{\mathbf{s}, \mathbf{d}, \top, \bot\}, \leq)$ be a poset, where $\leq$ is defined as: $\bot \leq \mathbf{s} \leq \top$ and $\bot \leq \mathbf{d} \leq \top$. $(B^Q, \leq)$ is a poset, where $\leq$ is defined pointwise. Moreover, for any $\pi_1, \pi_2 \in \Pi^Q$, let $\pi_1 \leq \pi_2$ if π_1 rafines π_2, i.e. for every block $X \in \pi_1$ there exists a block $Y \in \pi_2$ such that $X \subseteq Y$. Finally, for any $(b_1, \pi), (b_2, \pi_2) \in \mathcal{A}^Q$, $(b_1, \pi) \leq (b_2, \pi_2)$ if $b_1 \leq b_2$ and $\pi_1 \leq \pi_2$.

Proposition 1. *For any finite set Q, $(\mathcal{A}^Q, \leq)$ is a complete partial order, with $\bot = (\lambda q.\bot, \{\{q\}, q \in Q\})$ as least element.*

Proof. Every chain has a supremum since Q is finite. □

Basic operations of meet and join are defined on $\mathcal{A}^Q$. It turns out that contrary to $\mathcal{D}^Q$, $\langle \mathcal{A}^Q, \vee, \wedge, \bot, (\lambda q.\top, \{Q\}) \rangle$ is a lattice.

A removal operation on partitions is introduced as follows: for a given partition $\pi = \{Q_i, i \in I\}$, let $\pi \setminus q = \{Q_i \setminus \{q\}, i \in I\} \cup \{\{q\}\}$. Moreover, for any pair of qubits $q_1, q_2 \in Q$, let $[q_1, q_2] = \{\{q \mid q \in Q \setminus \{q_1, q_2\}\}, \{q_1, q_2\}\}$.

Finally, for any $b \in B^Q$, any $q_0, q \in Q$, any $k \in \{\mathbf{s}, \mathbf{d}, \top, \bot\}$, let

$$b_q^{q_0 \mapsto k} = \begin{cases} k & \text{if } q = q_0 \\ b_q & \text{otherwise} \end{cases}$$

We are now ready to define the abstract semantics of the language:

Definition 6 (Denotational abstract semantics). *For any program $\langle C, Q\rangle$, let $[\![C]\!]^\natural : \mathcal{A}^Q \to \mathcal{A}^Q$ be defined as follows: For any $(b, \pi) \in \mathcal{A}^Q$,*

$$[\![\mathsf{skip}]\!]^\natural(b,\pi) = (b,\pi)$$

$$[\![C_1; C_2]\!]^\natural(b,\pi) = [\![C_2]\!]^\natural \circ [\![C_1]\!]^\natural(b,\pi)$$

$$[\![\sigma(q)]\!]^\natural(b,\pi) = (b,\pi)$$

$$\begin{aligned}[\![\mathsf{H}(q)]\!]^\natural(b,\pi) &= (b^{q\mapsto \mathbf{d}}, \pi) \textit{ if } b_q = \mathbf{s} \\ &= (b^{q\mapsto \mathbf{s}}, \pi) \textit{ if } b_q = \mathbf{d} \\ &= (b,\pi) \textit{ otherwise}\end{aligned}$$

$$\begin{aligned}[\![\mathsf{T}(q)]\!]^\natural(b,\pi) &= (b^{q\mapsto \top}, \pi) \textit{ if } b_q = \mathbf{d} \\ &= (b^{q\mapsto \mathbf{s}}, \pi) \textit{ if } b_q = \bot \\ &= (b,\pi) \textit{ otherwise}\end{aligned}$$

$$\begin{aligned}[\![\mathsf{CNot}(q_1,q_2)]\!]^\natural(b,\pi) &= (b,\pi) \quad \textit{if } b_{q_1} = \mathbf{s} \textit{ or } b_{q_2} = \mathbf{d} \\ &= (b^{q_1\mapsto \mathbf{s}}, \pi) \quad \textit{if } b_{q_1} = \bot \textit{ and } b_{q_2} > \bot \\ &= (b^{q_2\mapsto \mathbf{d}}, \pi) \quad \textit{if } b_{q_1} > \bot \textit{ and } b_{q_2} = \bot \\ &= (b^{q_1\mapsto \mathbf{s}, q_2\mapsto \mathbf{d}}, \pi) \quad \textit{if } b_{q_1} = \bot \textit{ and } b_{q_2} = \bot \\ &= (b^{q_1,q_2\mapsto \top}, \pi \vee [q_1,q_2]) \quad \textit{otherwise}\end{aligned}$$

$$[\![\mathsf{if}\ q\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2\ ]\!]^\natural(b,\pi) = \big([\![C_1]\!]^\natural(b^{q\mapsto \mathbf{s}}, \pi\setminus q) \vee [\![C_2]\!]^\natural(b^{q\mapsto \mathbf{s}}, \pi\setminus q)\big)$$

$$\begin{aligned}[\![\mathsf{while}\ q\ \mathsf{do}\ C\ ]\!]^\natural(b,\pi) &= \mathsf{lfp}\left(\lambda f.\lambda\pi.\big(f \circ [\![C]\!]^\natural(b^{q\mapsto \mathbf{s}},\pi\setminus q) \vee (b^{q\mapsto \mathbf{s}}, \pi\setminus q)\big)\right) \\ &= \bigvee\nolimits_{n\in\mathbb{N}} \left(F_q^\natural \circ ([\![C]\!]^\natural \circ F_q^\natural)^n\right)\end{aligned}$$

where $F_q^\natural = \lambda(b,\pi).(b^{q\mapsto \mathbf{s}}, \pi\setminus q)$.

Intuitively, quantum operations act on entanglement as follows:

- A 1-qubit measurement makes the measured qubit separable from the rest of the memory. Moreover, the state of the measured qubit is in the standard basis.
- A 1-qubit unitary transformation does not modify entanglement. Any Pauli operator $\sigma \in \{\sigma_x, \sigma_y, \sigma_z\}$ preserves the standard and the diagonal basis of

the qubits. Hadamard H transforms a state of the standard basis into a state of the diagonal basis and vice-versa. Finally the phase T preserves the standard basis but not the diagonal basis.

– The 2-qubit unitary transformation $CNot$, applied on q_1 and q_2 may create entanglement between the qubits or not. It turns out that if q_1 is in the standard basis, or q_2 is in the diagonal basis, then no entanglement is created and the basis of q_1 and q_2 are preserved. Otherwise, since a sound approximation is desired, $CNot$ is abstracted into an operation which creates entanglement.

Remark 1. Notice that the space needed to store a partition of n elements is $O(n)$. Moreover, meet, join and removal and can be done in either constant or linear time.

Example 3. The abstract semantics of the teleportation (see example 1) is $[\![\mathsf{teleportation}]\!]^\natural : \mathcal{A}^{\{q_1,q_2,q_3\}} \to \mathcal{A}^{\{q_1,q_2,q_3\}} = \lambda(b,\pi).(b^{q_1,q_2\mapsto\mathbf{s},q_3\mapsto\top},\bot)$. Thus, for any 3-qubit state, the state of the memory after the teleportation is fully separable.

Assume that a fourth qubit q_4 is entangled with q_1 before the teleportation, whereas q_2 and q_3 are in the state $\mathrm{P}^{\mathsf{true}}$. So that, the state of the memory before the teleportation is $[q_1,q_4]$-separable. The abstract semantics of $\langle\mathsf{teleportation},\{q_1,q_2,q_3,q_4\}\rangle$ is such that

$$[\![\mathsf{teleportation}]\!]^\natural(b,[q_1,q_4]) = (b^{q_1,q_2\mapsto\mathbf{s},q_3\mapsto\top},[q_3,q_4])$$

Thus the abstract semantics predicts that q_3 is entangled with q_4 at the end of the teleportation, even if q_3 never interacts with q_4.

Example 4. Consider the program $\langle\mathsf{trap},\{q_1,q_2\}\rangle$, where

$$\mathsf{trap} = \mathsf{CNot}(q_1,q_2);\mathsf{CNot}(q_1,q_2)$$

Since $CNot$ is self-inverse, $[\![\mathsf{trap}]\!] : \mathcal{D}^{\{q_1,q_2\}} \to \mathcal{D}^{\{q_1,q_2\}} = \lambda\rho.\rho$. For instance, $[\![\mathsf{trap}]\!](\frac{1}{2}(P^{\mathsf{true}}+P^{\mathsf{false}})\otimes P^{\mathsf{true}}) = \frac{1}{2}(P^{\mathsf{true}}+P^{\mathsf{false}})\otimes P^{\mathsf{true}}$.

However, if $b_{q_1} = \mathbf{d}$ and $b_{q_2} = \mathbf{s}$ then

$$[\![\mathsf{trap}]\!]^\natural(b,\{\{q_1\},\{q_2\}\}) = (b^{q_1\mapsto\top,q_1\mapsto\top},\{\{q_1,q_2\}\})$$

Thus, according to the abstract semantics, at the end of the computation, q_1 and q_2 are entangled.

4.2 Soundness

Example 4 points out that the abstract semantics is an approximation, so it may differ from the entanglement evolution of the concrete semantics. However, in this section, we prove the soundness of the abstract interpretation (theorem 1).

First, we define a function $\beta : \mathcal{D}^Q \to B^Q$ such that $\beta(\rho)$ describes which qubits of ρ are in the standard or diagonal basis:

Definition 7. *For any finite Q, let $\beta : \mathcal{D}^Q \to B^Q$ such that for any $\rho \in \mathcal{D}^Q$, and any $q \in Q$,*

$$\beta(\rho)_q = \begin{cases} \mathbf{s} & \textit{if } P_q^{\mathsf{true}}\rho P_q^{\mathsf{false}} = P_q^{\mathsf{false}}\rho P_q^{\mathsf{true}} = 0 \\ \mathbf{d} & \textit{if } (P_q^{\mathsf{true}} + P_q^{\mathsf{false}})\rho(P_q^{\mathsf{true}} - P_q^{\mathsf{false}}) = (P_q^{\mathsf{true}} - P_q^{\mathsf{false}})\rho(P_q^{\mathsf{true}} + P_q^{\mathsf{false}}) = 0 \\ \top & \textit{otherwise} \end{cases}$$

A natural soundness relation is then:

Definition 8 (Soundness relation). *For any finite set Q, let $\sigma \in \wp(\mathcal{D}^Q, \mathcal{A}^Q)$ be the soundness relation:*

$$\sigma = \{(\rho, (b, \pi)) \mid \rho \textit{ is } \pi\textit{-separable and } \beta(\rho) \le b\}$$

The approximation relation is nothing but the partial order $\le$: (b, π) is a more precise approximation than (b', π') if $(b, \pi) \le (b', \pi')$. Notice that the abstract soundness assumption is satisfied: if ρ is π-separable and $\pi \le \pi'$ then ρ is π'-separable. So, $(\rho, a) \in \sigma$ and $(\rho, a) \le (\rho', a')$ imply $(\rho', a') \in \sigma$.

However, the best approximation is not ensured. Indeed, there exist some 3-qubit states [6,4] which are separable according to any of the 3 bipartitions of their qubits $\{a, b, c\}$ but which are not $\{\{a\}, \{b\}, \{c\}\}$-separable. Thus, the best approximation does not exist.

However, the soundness relation σ satisfies the following lemma:

Lemma 1. *For any finite set Q, any $\rho_1, \rho_2 \in \mathcal{D}^Q$, and any $a_1, a_2 \in \mathcal{A}^Q$,*

$$(\rho_1, a_1), (\rho_2, a_2) \in \sigma \implies (\rho_1 + \rho_2, \pi_1 \vee \pi_2) \in \sigma$$

Moreover, the abstract semantics is monotonic according to the approximation relation:

Lemma 2. *For any command C, $[\![C]\!]^\natural$ is $\le$-monotonic: for any $\pi_1, \pi_2 \in \mathcal{A}^Q$,*

$$\pi_1 \le \pi_2 \implies [\![C]\!]^\natural(\pi_1) \le [\![C]\!]^\natural(\pi_2)$$

Proof. The proof is by induction on C.

Theorem 1 (Soundness). *For any program $\langle C, Q\rangle$, any $\rho \in \mathcal{D}^Q$, and any $a \in \mathcal{A}^Q$,*

$$(\rho, a) \in \sigma \implies ([\![C]\!](\rho), [\![C]\!]^\natural(a)) \in \sigma$$

Proof. The proof is by induction on C.

In other words, if ρ is π-separable and $\beta(\rho) \le b$, then $[\![C]\!](\rho)$ is π'-separable and $\beta([\![C]\!](\rho)) \le b'$, where $(b', \pi') = [\![C]\!]^\natural(b, \pi)$.

5 Conclusion and Perspectives

In this paper, we have introduced the first quantum entanglement analysis based on abstract interpretation. Since a classical domain for driving a sound and complete analysis of entanglement is exponentially large in the number of qubits, an abstract domain based on partitions has been introduced. Moreover, since the concrete domain of superoperators is not a lattice, no Galois connection can be established between concrete and abstract domains. However, despite the absence of best abstraction, the soundness of the entanglement analysis has been proved.

The abstract domain is not only composed of partitions of the memory, but also of descriptions of the qubits which are in a basis state according to the standard or diagonal basis. Thanks to this additional information, the entanglement analysis is more subtle than an analysis of interactions: the $CNot$ transformation is not an entangling operation if the first qubit is in the standard basis or if the second qubit is in the diagonal basis.

A perspective, in order to reach a more precise entanglement analysis, is to introduce a more concrete abstract domain, adding for instance a third basis, since it is known that there are three mutually unbiased basis for each qubit.

A simple quantum imperative language is considered in this paper. This language is expressive enough to encode any quantum evolution. However, a perspective is to develop such abstract interpretation in a more general setting allowing high-order functions, representation of classical variables, or unbounded quantum memory. The objective is also to provide a practical tool for analysing entanglement evolution of more sophisticated programs, like Shor's algorithm for factorisation [19].

Another perspective is to consider that a quantum computer is available for driving the entanglement analysis. Notice that such an analysis of entanglement evolution is not trivial, even if a quantum computer is available, since a tomography [21] is required to know the entanglement of the quantum memory state[9].

Acknowledgements

The author would like to thank Philippe Jorrand for stimulating and insightful discussions, Samson Abramsky, Pablo Arrighi, Renan Fargetton, and Frédéric Prost for fruitful comments. The author is supported by EC STREP FP6-033763 Foundational Structures for Quantum Information and Computation (QICS.)

References

1. Abramsky, S.: A Cook's tour of a simple quantum programming language. In: 3rd International Symposium on Domain Theory, Xi'an, China (May 2004)
2. Aspect, A., Grangier, P., Roger, G.: Experimental tests of realistic local theories via Bell's theorem. Phys. Rev. Lett. 47, 460 (1981)

[9] It mainly means that in order to obtain an approximation of the quantum memory entanglement, several copies of the memory state are consumed.

3. Bennett, C.H., Brassard, G., Crépeau, C., Jozsa, R., Peres, A., Wootters, W.K.: Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. Phys. Rev. Lett. 70, 1895–1899 (1993)
4. Bennett, C.H., DiVincenzo, D.P., Mor, T., Shor, P.W., Smolin, J.A., Terhal, B.M.: Unextendible product bases and bound entanglement. Phys. Rev. Lett. 82, 53–85 (1999)
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
6. Eggeling, T., Werner, R.F.: Separability properties of tripartite states with uuu -symmetry. Phys. Rev. A 63(0421111) (2001)
7. Einstein, A., Podolsky, B., Rosen, N.: Can quantum-mechanical description of reality be considered complete? Phys. Rev. 47(10), 777–780 (1935)
8. Gay, S.J.: Quantum programming languages: Survey and bibliography. Mathematical Structures in Computer Science 16(4) (2006)
9. Gay, S.J., Rajagopal, A.K., Papanikolaou, N.: Qmc: A model qmc: A model checker for quantum systems. arxiv:0704.3705 (2007)
10. Gurvits, L.: Classical deterministic complexity of Edmonds' problem and quantum entanglement. In: Proceedings of the 35-th ACM Symposium on Theory of Computing, p. 10. ACM Press, New York (2003)
11. Jorrand, P., Mhalla, M.: Separability of pure n-qubit states: two characterizations. IJFCS 14(5), 797–814 (2003)
12. Kashefi, E.: Quantum domain theory - definitions and applications. In: Proceedings of Computability and Complexity in Analysis (CCA 2003) (2003)
13. Nielsen, M.A., Chuang, I.L.: Quantum computation and quantum information. Cambridge University Press, New York (2000)
14. Perdrix, S.: Formal models of quantum computation: resources, abstract machines and measurement-based quantum computation (in french). PhD thesis, Institut National Polytechnique de Grenoble (2006)
15. Perdrix, S.: A hierarchy of quantum semantics. In: The Proceedings of the 3rd International Workshop on Development of Computational Models (to appear, 2007)
16. Prost, F., Zerrari, C.: A logical analysis of entanglement and separability in quantum higher-order functions. arXiv.org:0801.0649 (2008)
17. Selinger, P.: A brief survey of quantum programming languages. In: Kameyama, Y., Stuckey, P.J. (eds.) FLOPS 2004. LNCS, vol. 2998, pp. 1–6. Springer, Heidelberg (2004)
18. Selinger, P.: Towards a quantum programming language. Mathematical Structures in Computer Science 14(4), 527–586 (2004)
19. Shor, P.: Algorithms for quantum computation: Discrete logarithms and factoring. In: Goldwasser, S. (ed.) Proceedings of the 35^{th} Annual Symposium on Foundations of Computer Science, pp. 124–134. IEEE Computer Society Press, Los Alamitos (1994)
20. Van den Nest, M., Miyake, A., Dür, W., Briegel, H.J.: Universal resources for measurement–based quantum computation (2006)
21. White, A.G., Gilchrist, A., Pryde, G.J., O'Brien, J.L., Bremner, M.J., Langford, N.K.: Measuring two-qubit gates. J. Opt. Soc. Am. B 24(2), 172–183 (2007)

Language Strength Reduction*

Nicholas Kidd[1], Akash Lal[1,**], and Thomas Reps[1,2]

[1] University of Wisconsin, Madison, WI, USA
{kidd,akash,reps}@cs.wisc.edu
[2] GrammaTech, Inc., Ithaca, NY, USA

Abstract. This paper concerns methods to check for atomic-set serializability violations in concurrent Java programs. The straightforward way to encode a reentrant lock is to model it with a context-free language to track the number of successive lock acquisitions. We present a construction that replaces the context-free language that describes a reentrant lock by a regular language that describes a non-reentrant lock. We call this replacement *language strength reduction.* Language strength reduction produces an average speedup (geometric mean) of 3.4. Moreover, for 2 programs that previously exhausted available space, the tool is now able to run to completion.

1 Introduction

Vaziri et al. [1] define an *atomic set* as a set of memory locations that share a consistency property, and a *unit-of-work* as a code fragment that preserves the consistency property. They specify eleven forbidden *data-access patterns* on atomic sets; and show that an atomic-set serializability violation occurs iff one of the data-access patterns is observed during a unit-of-work.

Empire [2] is a static violation[1] detector for Java. Empire abstracts a concurrent Java program into a program written in the Empire Modeling Language (EML). An EML program consists of a finite set of processes, a finite set of global variables, and a finite set of locks. Each process consists of a set of (recursive) functions with the standard control operators. As in Java, an EML lock is reentrant, i.e., it can be acquired multiple times by the process that owns the lock, but it also must be successively released the same number of times. An EML lock is acquired and released by entering and exiting, respectively, a function that is synchronized on the lock. (Java synchronized blocks are modeled as inlined anonymous function invocations in EML.) EML provides a `unit` block that denotes a unit-of-work. The `unit` blocks are allowed to be nested. An example EML process is given in Fig. 1.

An execution trace of an EML process is described by a string of actions, where an action corresponds to reading (writing) a variable, acquiring (releasing) a lock,

* Supported by NSF under grants CCF-0540955 and CCF-0524051.
** Supported by a Microsoft Research Fellowship.

[1] For this paper, the term "violation" means "atomic-set serializability violation".

© Springer-Verlag Berlin Heidelberg 2008

```
lock: l;
var : v;

process P0 {
    synchronized(l) get { read v; }

    synchronized(l) set { write v; }

    synchronized(l) testAndSet {
        get();
        if( * )
            set();
    }

    main {
        unit {
            testAndSet()
        }
    }
}
```

Fig. 1. Example program that makes use of reentrant locking

or entering (exiting) a `unit` block. The set of all execution traces of an EML process is described by a context-free language (CFL) of actions. Similarly, the set of all behaviors of an EML lock is described by a CFL. Finally, the set of all interleaved execution traces of an EML program is described by the intersection of a set S of CFLs—one for each EML process and one for each EML lock. Intersection ensures that the mutual-exclusion property of locks is obeyed.

Violation detection is performed by determining the emptiness of the intersection of the set S, augmented with a regular language L_{data} that defines a data-access pattern. Determining the emptiness of the intersection of two or more CFLs is undecidable. This issue is addressed by translating the EML program, along with L_{data}, into a communicating pushdown system (CPDS) [3,4], for which a semi-decision procedure is implemented in the CPDS model checker [4]. The semi-decision procedure over-approximates each CFL by a regular language and then checks whether the intersection of the regular languages is empty (which is decidable). If the intersection is empty or it contains a valid string in each of the original CFLs, then the model checker terminates. Otherwise, the process is repeated using a tighter regular over-approximation of each CFL.

If a language is known to be regular (e.g., L_{data}), then the CPDS model checker can be directed to treat it as such (determining if a CFL is regular is also undecidable). This has two key advantages:

1. *Precision* increases because the model checker uses the exact language.
2. *Cost* decreases because the model checker avoids approximating a CFL.

This paper presents a generic technique that we use to reduce the number of CFLs necessary to model an EML program. It is based on the observation that the CFL for an EML lock can be replaced by a regular language because the EML lock's acquisitions and releases are synchronized with function calls and returns. We call the process of replacing a CFL by a regular language *language* *strength reduction*. For an EML program with m processes and n locks, applying language strength reduction allows the program to be described by m CFLs and n regular languages, versus $m + n$ CFLs.

Contributions. The observation that pushdown automata are closed under intersection when the stacks are synchronized was formalized in the work of Alur and Madhusudan [5,6]. They defined nested-word languages, which make stack operations explicit in the words of the language, and nested-word automata (NWA), which accept such languages. They showed that these languages are closed under intersection. However, their result does not apply in our setting.

In our setting, a CPDS consists of a set of extended weighted pushdown systems (EWPDSs) [7]. EWPDSs are a generic formalism for modeling recursive programs (cf. §3.1). They are able to model more powerful program abstractions than the pushdown automata used in [5,6]. (EWPDSs can model infinite-state data abstractions, as opposed to pushdown automata, which can only model finite-state data abstractions.) EWPDSs allow one to compute meet-over-all-valid-paths (MOVP) values for the abstraction, which goes beyond the capabilities of the approach proposed by Alur and Madhusudan. The MOVP values capture the set of behaviors of the program modeled by the EWPDS.

Our approach is similar in spirit to [6]. We use an NWA A to model the locking behavior of an EML process. We define the nested-word language of an EWPDS (cf. §4.1) by associating a nested-word with every path of the EWPDS. This makes its stack operations explicit. We give a generic construction that combines A with an EWPDS $\mathcal{E}$ to produce another EWPDS $\mathcal{E}_A$ whose nested-word language is the intersection of the nested-word languages of $\mathcal{E}$ and A. Computing the MOVP value over $\mathcal{E}_A$ captures the set of all behaviors of the program modeled by $\mathcal{E}$ that respect the locking behaviors described by A.

The key to language strength reduction is distinguishing between the lock acquisitions and releases that change the owner of a lock l and those that do not. We show how to achieve this using the NWA A. We then transfer this ability to the EWPDS $\mathcal{E}$ for an EML process via the construction of $\mathcal{E}_A$. This enables us to perform language strength reduction for the lock l (cf. §5).

Our work makes the following contributions:

– We define the notion of the nested-word language of an EWPDS (§4.1). We give a construction to combine an NWA A with an EWPDS $\mathcal{E}$ to produce another EWPDS $\mathcal{E}_A$ (§4.2). This generalizes previous results, and permits verification to be performed using a broader class of abstract domains (see Defn. 2).
– We show how the construction allows one to perform language strength reduction (§5).
– We analyzed 5 programs from the concurrency testing benchmark suite by Eytani et al. [8]. Our technique obtained an average speedup of 3.4 on 3 of the programs. Moreover, for the 2 programs that previously exhausted available space, the tool is now able to run to completion.

$$\begin{array}{lll}
S & \to U & \\
M & \to \epsilon & \mid M\ M \mid (\ M\ ) \\
U & \to M & \mid M\ U \mid (\ U
\end{array}
\qquad
\begin{array}{llll}
S & \to U^o & & \\
M^o & \to \epsilon & \mid M^o\ M^o & \mid (_o\ M^n\ )_o \\
U^o & \to M^o & \mid M^o\ U^o & \mid (_o\ U^n \\
M^n & \to \epsilon & \mid M^n\ M^n & \mid (_n\ M^n\ )_n \\
U^n & \to M^n & \mid M^n\ U^n & \mid (_n\ U^n
\end{array}
\qquad
S \to (_o\ )_o\ S \mid (_o \mid \epsilon$$

(a) (b) (c)

Fig. 2. (a) Grammar for the CFL of a reentrant lock. (b) Grammar that distinguishes between outermost and nested parentheses. (c) Grammar for the regular language of a non-reentrant lock.

The remainder of the paper is organized as follows: §2 provides an overview. §3 presents definitions and examples. §4 presents the nested-word language of an EWPDS and the construction that combines an NWA with an EWPDS. §5 presents the language-strength-reduction transformation. §6 describes our experiments. §7 discusses related work.

2 Overview

Consider the EML process in Fig. 1. Let "(" and ")" denote entering and exiting a `synchronized(l)` function, "[" and "]" denote entering and exiting a `unit` block, and R_v and W_v denote reading and writing to the variable v, respectively. The program path

Path 1: *main* → *testAndSet* → *get* → *set* → *testAndSet* → *main*

can be described by the word w_{path} = "$[((R_v)(W_v))]$". Removing all symbols that do not model a change in the state of the lock l produces the word w_l = "$(()())$". In general, due to recursion, the language that describes the set of possible program behaviors with respect to l is a partially-balanced matched-parenthesis language, whose grammar is shown in Fig. 2(a).

For *Path 1*, there are two distinct types of lock acquisitions: ownership-changing acquisitions (OC) and non-ownership-changing acquisitions (nOC). The dual also holds for lock releases. With respect to w_l, these two distinct types correspond to outermost parentheses, denoted by "$(_o)_o$", and nested parentheses, denoted by "$(_n)_n$", respectively. Using this notation, w_l can be rewritten as "$(_o(_n)_n(_n)_n)_o$". Fig. 2(b) extends this to the language level by distinguishing between the outermost and nested parentheses of Fig. 2(a).

Observation 1. *With respect to the executions of an EML program, only the OC lock acquisitions and releases enforce mutual exclusion. For a program trace, projecting out the nOC lock acquisitions and releases does not change the set of instructions that are guarded by locks.*

Projecting out the nested parentheses for w_l results in "$(_o)_o$". Performing the projection on the grammar in Fig. 2(b) results in a regular language whose grammar is shown in Fig. 2(c).

This paper presents a technique that allows us to use the simpler language in Fig. 2(c) in place of the language in Fig. 2(a). We call this replacement *language strength reduction*. Language strength reduction provides the precision and cost benefits highlighted by items 1 and 2 of §1.

Language strength reduction relies on the ability to distinguish between the OC and nOC lock acquisitions of an EML process. In §3.3, we show how this distinction can be captured by an NWA. Having defined the language of Fig. 2(b) via an NWA A, we combine it with the EWPDS $\mathcal{E}$ that represents an EML process. This results in another EWPDS $\mathcal{E}_A$ on which we then project out all nOC lock acquisitions and releases—the end result being that each EML lock is modeled by the regular language shown in Fig. 2(c) in the CPDS model checker. Using the simpler language of Fig. 2(c) leads to the speedups reported in §6.

The goal of language strength reduction is to model reentrant locks with non-reentrant locks without sacrificing soundness or precision. This problem can be tackled by either source-code modification or by manipulating the program model. In our model checker's tool chain, a CPDS is produced from a concurrent Java program, and thus we followed the approach of modifying the EWPDSs that make up the generated CPDS. A benefit of this approach is that we have developed generic techniques that apply to a declarative specification of the set of locks. That is, given the set of lock names, the techniques we present perform language strength reduction automatically.

3 Definitions and Examples

3.1 Extended Weighted Pushdown Systems

Definition 1. *A* ***pushdown system*** *(PDS) is a triple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of* states*, Γ is a finite set of stack symbols, and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is a finite set of rules. A* ***configuration*** *of $\mathcal{P}$ is a pair $\langle p, u\rangle$ where $p \in P$ and $u \in \Gamma^*$. A rule $r \in \Delta$ is written as $\langle p, \gamma\rangle \hookrightarrow \langle p', u\rangle$, where $p, p' \in P$, $\gamma \in \Gamma$, and $u \in \Gamma^*$. These rules define a transition relation $\Rightarrow$ on configurations of $\mathcal{P}$ as follows: if $r = \langle p, \gamma\rangle \hookrightarrow \langle p', u'\rangle$, then $\langle p, \gamma u\rangle \Rightarrow \langle p', u'u\rangle$ for all $u \in \Gamma^*$. The reflexive transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$.*

Without loss of generality, we restrict PDS rules to have at most two stack symbols on the right-hand side [9]. A rule $r = \langle p, \gamma\rangle \hookrightarrow \langle p', u\rangle$, $u \in \Gamma^*$, is called a *push*, *step*, or *pop* rule if $|u| = 2$, $|u| = 1$, or $|u| = 0$, respectively.

A PDS naturally models a program's control flow. The standard approach is as follows: P contains a single state p, Γ corresponds to the nodes of the program's interprocedural control flow graph (ICFG), and Δ corresponds to edges of the program's ICFG (see Fig. 3). We denote the entry point of a program's main function by e_{main}, and let $c_{\text{init}} = \langle p, e_{\text{main}}\rangle$. A *run* of $\mathcal{P}$ is a rule sequence $\rho = [r_1, \ldots, r_j]$ that transforms c_{init} into some other configuration c.[2] We denote the

[2] It is not necessary to restrict the definition of a run to start from the initial configuration. However, this simplifies the discussion.

Rule	Control flow modeled
$\langle p, n_1 \rangle \hookrightarrow \langle p, n_2 \rangle$	Intraprocedural edge $n_1 \rightarrow n_2$
$\langle p, n_c \rangle \hookrightarrow \langle p, e_f\ r_c \rangle$	Call to f, with entry e_f, from n_c that returns to r_c
$\langle p, x_f \rangle \hookrightarrow \langle p, \varepsilon \rangle$	Return from f at exit x_f

Fig. 3. The encoding of an ICFG's edges as PDS rules

set of all runs of $\mathcal{P}$ by $Runs(\mathcal{P})$, which represents the set of all interprocedurally-valid paths in the program.

An extended weighted pushdown system (EWPDS) is obtained by augmenting a PDS with a weight domain [10,3] and a set of merging functions [7]. Weights encode the effect that each statement (or PDS rule) has on the data state of the program. Merging functions are used to fuse the local state of the calling procedure as it existed just before the call with the global state produced by the called procedure.

Definition 2. *A* **weight domain** *is a tuple* $(D, \oplus, \otimes, \overline{0}, \overline{1})$*, where* D *is a set whose elements are called* **weights***,* $\overline{0}, \overline{1} \in D$*, and* $\oplus$ *(the combine operation) and* $\otimes$ *(the extend operation) are binary operators on* D *such that*

1. $(D, \oplus)$ *is a commutative monoid with* $\overline{0}$ *as its neutral element, and where* $\oplus$ *is idempotent (i.e., for all* $a \in D$*,* $a \oplus a = a$*).* $(D, \otimes)$ *is a monoid with the neutral element* $\overline{1}$*.*
2. $\otimes$ *distributes over* $\oplus$*, i.e., for all* $a, b, c \in D$ *we have*
$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \text{ and } (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)\ .$$
3. $\overline{0}$ *is an annihilator with respect to* $\otimes$*, i.e., for all* $a \in D$*,* $a \otimes \overline{0} = \overline{0} = \overline{0} \otimes a$*.*
4. *In the partial order* $\sqsubseteq$ *defined by* $\forall a, b \in D,\ a \sqsubseteq b$ *iff* $a \oplus b = a$*, there are no infinite descending chains.*

Example: The Prefix Weight Domain for CPDSs [3]. For a CFL L over finite alphabet Σ, the prefix abstraction precisely models each word $w \in L$ whose length is less than a bound k. If $|w| \geq k$, then w is approximated by the regular language $w|_k \Sigma^*$, where $w|_k$ denotes the prefix of w of length k. Because there are only a finite number of words and prefixes whose lengths are less than or equal to k, the prefix abstraction produces a regular approximation of L.

For two words $w_1 = a_1 \ldots a_i$ and $w_2 = b_1 \ldots b_j$, let $w_1 \bowtie_k w_2$ be the word $(a_1 \ldots a_i b_1 \ldots b_j)|_k$. We extend $\bowtie_k$ to finite sets in the obvious way. For a finite alphabet Σ and bound k, let D be the powerset of $\bigcup_{0 \leq i \leq k} \Sigma^i$. The prefix weight domain is defined as $\mathcal{S}|_k = (D, \cup, \bowtie_k, \emptyset, \{\epsilon\})$.

Definition 3. *A function* $m : D \times D \rightarrow D$ *is a* **merging function** *with respect to a weight domain* $(D, \oplus, \otimes, \overline{0}, \overline{1})$ *if it satisfies the following properties:*

1. **Strictness.** *For all* $a \in D$*,* $m(\overline{0}, a) = m(a, \overline{0}) = \overline{0}$*.*
2. **Distributivity.** *The function distributes over* $\oplus$*. For all* $a, b, c \in D$*,*
$$m(a \oplus b, c) = m(a, c) \oplus m(b, c) \text{ and } m(a, b \oplus c) = m(a, b) \oplus m(a, c)$$

	Rules	Weight	d_{const}
1	$\langle p, e_{\text{main}}\rangle \hookrightarrow \langle p, n_{16}\rangle$	$\overline{1}$	
2	$\langle p, n_{16}\rangle \hookrightarrow \langle p, n_{17}\rangle$	$\{ [\}$	
3	$\langle p, n_{17}\rangle \hookrightarrow \langle p, e_{\text{testAndSet}}\ n_{18}\rangle$	$\{ (\}$	$\{) \}$
4	$\langle p, e_{\text{testAndSet}}\rangle \hookrightarrow \langle p, n_{10}\rangle$	$\overline{1}$	
5	$\langle p, n_{10}\rangle \hookrightarrow \langle p, e_{\text{get}}\ n_{11}\rangle$	$\{ (\}$	$\{) \}$
6	$\langle p, e_{\text{get}}\rangle \hookrightarrow \langle p, x_{\text{get}}\rangle$	$\{R_v\}$	
7	$\langle p, x_{\text{get}}\rangle \hookrightarrow \langle p, \epsilon\rangle$	$\overline{1}$	
8	$\langle p, n_{11}\rangle \hookrightarrow \langle p, n_{12}\rangle$	$\overline{1}$	
9	$\langle p, n_{12}\rangle \hookrightarrow \langle p, e_{\text{set}}\ x_{\text{testAndSet}}\rangle$	$\{ (\}$	$\{) \}$
10	$\langle p, e_{\text{set}}\rangle \hookrightarrow \langle p, x_{\text{set}}\rangle$	$\{W_v\}$	
11	$\langle p, x_{\text{set}}\rangle \hookrightarrow \langle p, \epsilon\rangle$	$\overline{1}$	
12	$\langle p, x_{\text{testAndSet}}\rangle \hookrightarrow \langle p, \epsilon\rangle$	$\overline{1}$	
13	$\langle p, n_{18}\rangle \hookrightarrow \langle p, x_{\text{main}}\rangle$	$\{] \}$	
14	$\langle p, x_{\text{main}}\rangle \hookrightarrow \langle p, \epsilon\rangle$	$\overline{1}$	
15	$\langle p, n_{11}\rangle \hookrightarrow \langle p, x_{\text{testAndSet}}\rangle$	$\overline{1}$	

Fig. 4. EWPDS rules that encode EML process `P0` from Fig. 1 (subscripts correspond to the line numbers). Only the constant weight d_{const} is shown for the merging functions.

Example: The Prefix Merging Functions of Empire. The prefix merging functions used by Empire are of the form $\lambda d_1.\lambda d_2.d_1 \otimes d_2 \otimes d_{\text{const}}$, where d_{const} is either $\overline{1}$ for invoking non-synchronized functions, or $\{) \}$ for invoking a function that is synchronized on a lock l. Note that placing the close-parenthesis symbol, corresponding to the release of a lock, inside of a merge function accurately reflects the behavior of returning from a synchronized function.

Definition 4. *Let $\mathcal{M}$ be the set of all merging functions on weight domain $\mathcal{S}$, and let Δ_2 denote the set of push rules of a PDS $\mathcal{P}$. An* **extended weighted pushdown system** *is a quadruple $\mathcal{E} = (\mathcal{P}, \mathcal{S}, f, g)$ where $\mathcal{P} = (P, \Gamma, \Delta)$ is a PDS, $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$ is a weight domain, $f : \Delta \to D$ is a map that assigns a weight to each rule of $\mathcal{P}$, and $g : \Delta_2 \to \mathcal{M}$ assigns a merging function to each rule in Δ_2.*

Example: An EWPDS for an EML process. For an EML process π, an EWPDS $\mathcal{E}\langle\pi\rangle$ is generated using the schema from Fig. 3, the prefix weight domain, and the prefix merging functions. Fig. 4 presents the rules that encode process `P0` from Fig. 1.

Run of an EWPDS. A run of an EWPDS $\mathcal{E}$ is simply a run of its underlying PDS. We denote the set of all runs of $\mathcal{E}$ by $\mathit{Runs}(\mathcal{E})$, and the set of runs ending in configuration c as $\mathit{Runs}(\mathcal{E}, c)$. Using f and g, we can associate a value to a run ρ, denoted by $\mathit{val}(\rho)$. To do so, we define the helper functions $\mathit{val}[r]$, *build*, and *flatten*. The function $\mathit{val}[r](z, S)$ takes a weight and a weight-rule stack, and returns a weight and weight-rule stack:

$$\mathit{val}[r](z,S) = \begin{cases} (z \otimes f(r), S) & \text{if } r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\rangle \\ (\overline{1}, (z,r)||S) & \text{if } r = \langle p,\gamma\rangle \hookrightarrow \langle p',\gamma'\gamma''\rangle \\ (g(r_c)(z_c, f(r_c) \otimes z \otimes f(r)), S') & \text{if } r = \langle p,\gamma\rangle \hookrightarrow \langle p',\epsilon\rangle \\ & \quad \text{and } S = (z_c, r_c)||S' \\ (z \otimes f(r), S) & \text{if } r = \langle p,\gamma\rangle \hookrightarrow \rangle p', \epsilon \text{ and } S = \emptyset \end{cases}$$

The function $\mathit{build}(\rho)$ maps a run to a weight and weight-rule stack as follows:

$$\begin{aligned} \mathit{build}([]) &= (\overline{1}, \emptyset) \\ \mathit{build}([r_1, \ldots, r_j]) &= \mathit{val}[r_j](\mathit{build}([r_1, \ldots, r_{j-1}])) \end{aligned}$$

The function $\mathit{flatten}(z, S)$ "flattens" a weight and weight-rule stack by using the extend ($\otimes$) operation:

$$\begin{aligned} \mathit{flatten}(z, \emptyset) &= z \\ \mathit{flatten}(z, (z_c, r_c)||S') &= \mathit{flatten}(z_c \otimes f(r_c) \otimes z, S') \end{aligned}$$

Given these definitions, $\mathit{val}(\rho) = \mathit{flatten}(\mathit{build}(\rho))$.

Example: Valuation of Path 1. Using the EWPDS rules of Fig. 4, and for a prefix bound $k > 10$, one can verify that $\mathit{val}([r_1, \ldots, r_{14}]) = \{\ [((R_v))(W_v))]\ \}$, which is the set containing only the word given in §2 for *Path 1*.

Definition 5. *For EWPDS $\mathcal{E}$ and a set of configurations C, the* ***meet-over-all-valid-paths*** *value* $\mathrm{MOVP}_{\mathcal{E}}(C)$ *is defined as* $\bigoplus\{\mathit{val}(\rho) \mid \rho \in \mathit{Runs}(\mathcal{E}, c), c \in C\}$.

The MOVP value captures the net effect of all paths leading to a set of configurations. An algorithm for computing MOVP is given in [7].

Example: MOVP **for EML process** `P0` **from Fig. 1.** Let $\mathcal{E}\langle\mathtt{P0}\rangle$ be the EWPDS for process `P0` with rules given in Fig. 4. For a prefix bound $k > 10$, $\mathrm{MOVP}_{\mathcal{E}\langle\mathtt{P0}\rangle}(\langle p, x_{\mathrm{main}}\rangle) = \{\ [((R_v)(W_v))]\ ,\ [((R_v))]\ \}$. The first string describes the path that follows the true branch of the `if` statement at line 11 in Fig. 1, and the second string describes the path that follows the false branch. Because process `P0` has only two valid paths and $k > 10$, the MOVP weight precisely describes the behavior of process `P0`. However, if k was instead the value 8, then the result of the same MOVP computation would be $\{\ [((R_v)(W_v)\Sigma^*\ ,\ [((R_v))]\ \}$. Note that the first string has been approximated by an infinite set of strings.

3.2 Communicating Pushdown System

A CPDS consists of a set of EWPDSs $\mathcal{E}_1, \ldots, \mathcal{E}_n$, where each EWPDS $\mathcal{E}_i$ uses the prefix weight domain and merging functions, and a set of target configurations $C_1, \ldots, C_n$. The CPDS model checker computes: $S = \bigcap_{1 \le i \le n} \mathrm{MOVP}_{\mathcal{E}_i}(C_i)$. The set S is the intersection of the prefix abstractions for each CFL that is modeled by an EWPDS. If $S = \emptyset$, then so is the intersection of the CFLs. Otherwise, let w be the shortest word in S. If $|w| = k$, then k is incremented and the process repeats. Otherwise, w represents a concrete execution of the EML program that reaches the target configurations.

3.3 Nested Word Automata

Alur et al. [6] define a nested word to be a pair (w, v), where w is a word $a_1 \ldots a_k$ over a finite alphabet and v, the *nesting relation*, is a subset of $\{1, 2, \ldots, k\} \times$

$(\{1, 2, \ldots, k\} \cup \{\infty\})$. The nesting relation denotes a set of *properly nested* hierarchical edges of a nested word. For a valid nesting relation, $v(i, j)$ implies $i < j$, and for all i', j' such that $v(i', j')$ holds and $i < i'$, then either $j < i'$ or $j' < j$. Given v, i is a *call position* if $v(i, j)$ holds for some j, a *return position* if $v(k, i)$ holds for some k, and an *internal position* otherwise.

A set of nested words is *regular* if it can be modeled by a *nested-word automaton* (NWA) [6]. An NWA A is a tuple $(Q, \Sigma, q_0, \delta, F)$, where Q is a finite set of states, Σ is a finite alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, and δ is a transition relation that consists of three components:

- $\delta_c \subseteq Q \times \Sigma \times Q$ defines the transition relation for call positions.
- $\delta_i \subseteq Q \times \Sigma \times Q$ defines the transition relation for internal positions.
- $\delta_r \subseteq Q \times Q \times \Sigma \times Q$ defines the transition relation for return positions.

Starting from q_0, an NWA A reads a nested word $nw = (w, v)$ from left to right, and performs transitions (possibly non-deterministically) according to the input symbol and the nesting relation. That is, if A is in state q when reading input symbol σ at position i in w, then if i is a call or internal position, A makes a transition to q' using $(q, \sigma, q') \in \delta_c$ or $(q, \sigma, q') \in \delta_i$, respectively. Otherwise, i is a return position and $v(j, i)$ holds for some j. Let q_c be the state A was in just before the transition it made on the j^{th} symbol; then A uses $(q, q_c, \sigma, q') \in \delta_r$ to make a transition to q'. If, after reading nw, A is in a state $q \in F$, then A accepts nw [6].

We use $L(A)$ to denote the nested-word language that A accepts, and $L(A, q)$ to denote the nested-word language such that for each nested word $nw \in L(A, q)$, A is left in state q after reading nw. We extend this notion to sets of states in the obvious way. Thus, $L(A) = L(A, F)$.

Table 1. An NWA template for the locking behavior of an EML process

δ_c	δ_r	δ_i
$(q, e_{\text{sync}}, \boxtimes)$	$(\boxtimes, q_c, x_{\text{sync}}, q_c)$	(q, σ, q)
(q, e, q)	(q, q, x, q)	

An NWA Template for Lock Behavior. For an EML lock l and process π with set of functions *Sync* synchronized on l and set of functions *Fun* not synchronized on l, the locking behavior of π on l is defined by an NWA $A\langle\pi\rangle = (Q, \Sigma, q_0, \delta, F)$, where $Q = \{\boxtimes, \square\}$, Σ is the set of control locations of π, $q_0 = \square$, $F = Q$, and δ is defined in Tab. 1. (The transitions in Tab. 1 are instantiated for all $q \in Q$, $e_{\text{sync}} \in \{e_f \mid f \in \mathit{Sync}\}$, $e \in \{e_f \mid f \in \mathit{Fun}\}$, $x_{\text{sync}} \in \{x_f \mid f \in \mathit{Sync}\}$, $x \in \{x_f \mid f \in \mathit{Fun}\}$, and $\sigma \in (\Sigma - \{e_{\text{sync}}, x_{\text{sync}}, e, x\})$.)

$A\langle\pi\rangle$ consists of two states: locked ($\boxtimes$) and unlocked ($\square$). The entry to and exit from a function `f` are denoted by $e_{\mathtt{f}}$ and $x_{\mathtt{f}}$, respectively. When an l-synchronized function is called, $A\langle\pi\rangle$ makes a transition to the locked state via the transitions $(q, e_{\text{sync}}, \boxtimes)$. When returning from a function, the state of the caller is restored.

For example, the transitions $(\boxtimes, q_c, x_{\text{sync}}, q_c)$ ensure that $A\langle\pi\rangle$ goes to state q_c of the caller.

Template Usage. For EML process `P0` from Fig. 1, let $\mathcal{E}\langle\texttt{P0}\rangle$ be the EWPDS that models `P0` with the rules shown in Fig. 4, and let $A\langle\texttt{P0}\rangle$ be the NWA that results from instantiating the above template with `P0`. With respect to the locking behavior of `P0`, $\mathcal{E}\langle\texttt{P0}\rangle$ cannot distinguish between OC and nOC lock acquisitions and releases, while $A\langle\texttt{P0}\rangle$ is able to do so via its state space. The transitions $(\square, e_{\text{sync}}, \boxtimes)$ and $(\boxtimes, e_{\text{sync}}, \boxtimes)$ in δ_c are the OC and nOC lock acquisitions, respectively; and transitions $(\boxtimes, \boxtimes, e_{\text{sync}}, \boxtimes)$ and $(\boxtimes, \square, e_{\text{sync}}, \square)$ in δ_r are the nOC and OC lock releases, respectively.

We show how to combine $\mathcal{E}\langle\texttt{P0}\rangle$ and $A\langle\texttt{P0}\rangle$ in §4 to construct another EWPDS $\mathcal{E}_A\langle\texttt{P0}\rangle$, such that $\mathcal{E}_A\langle\texttt{P0}\rangle$ contains the same behaviors as $\mathcal{E}\langle\texttt{P0}\rangle$, but is able to distinguish between the OC and nOC lock acquisitions and releases. Once such a distinction can be made, we leverage *Observation 1* to remove all nOC lock acquisitions and releases from $\mathcal{E}_A\langle\texttt{P0}\rangle$. This makes it possible to model an EML lock with the trivial language shown in Fig. 2(c).

4 Combining an NWA with an EWPDS

We first define the notion of the *nested-word language* of an EWPDS, which establishes a relationship between the NWA and EWPDS formalisms. Additionally, it allows us to formally reason about the construction of §4.2 that combines an NWA with an EWPDS.

4.1 The Nested-Word Language of an EWPDS

The nested-word language of an EWPDS $\mathcal{E} = (\mathcal{P}, \mathcal{S}, f, g)$, denoted by $L(\mathcal{E})$, is defined in terms of the set of runs of $\mathcal{E}$. Intuitively, if (w, v) is a nested word in $L(\mathcal{E})$, w consists of the sequence of left-hand-side stack symbols $\gamma_1 \ldots \gamma_j$ for a run $[r_1, \ldots, r_j]$ of $\mathit{Runs}(\mathcal{E})$, and v encodes the matching calls and returns. We additionally require that the valuation of the run not be equal to the weight zero, i.e., $\mathit{val}(\rho) \neq \overline{0}$. This notion is formalized by defining the function *post*, which maps a run of $\mathcal{E}$ to a nested word. The function *post* is defined recursively in terms of the helper function $\mathit{post}[r](w, v)$.

For a nested word $nw = (w, v)$ and rule $r \in \Delta$, $\mathit{post}[r](w, v)$ is defined as follows:

$$\mathit{post}[r](w,v) = \begin{cases} (w\gamma, v) & \text{if } r = \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle \\ (w\gamma, (v - \{\langle i, \infty\rangle\}) \cup \{\langle i, |w\gamma|\rangle\}) & \text{if } r = \langle p, \gamma\rangle \hookrightarrow \langle p', \epsilon\rangle, \\ & \quad i = \mathit{max}(\{j \mid \langle j, \infty\rangle \in v\}) \\ (w\gamma, v \cup \{\langle |w\gamma|, \infty\rangle\}) & \text{if } r = \langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\ \gamma''\rangle \end{cases}$$

Using $post[r]$, we define the function $post([r_1 \ldots r_j])$[3] as follows:

$$\begin{aligned} post([]) &= (\epsilon, \emptyset) \\ post([r_1, \ldots, r_j]) &= post[r_j](post([r_1, \ldots, r_{j-1}])) \end{aligned}$$

Definition 6. *For an EWPDS $\mathcal{E}$, the nested-word language $L(\mathcal{E})$ is defined as $L(\mathcal{E}) = \{post(\rho) \mid \rho \in Runs(\mathcal{E}) \wedge val(\rho) \neq \overline{0}\}$.*

We will sometimes wish to further restrict $L(\mathcal{E})$ by an acceptance criterion, which we call *φ-acceptance.*

Definition 7. *The **φ-accepted** nested-word language for an EWPDS $\mathcal{E}$ and function $\varphi : D \rightarrow \mathbb{B}$ is defined as $L^{\varphi}(\mathcal{E}) = \{post(\rho) \mid \rho \in Runs(\mathcal{E}) \;\wedge\; val(\rho) \neq \overline{0} \wedge\; \varphi(val(\rho))\}$.*

4.2 Construction

The construction that combines an EWPDS $\mathcal{E}$ with an NWA A produces another EWPDS $\mathcal{E}_A$. The weight domain of $\mathcal{E}_A$ models the transition relation of A in addition to the original weight domain of $\mathcal{E}$. This is accomplished via a *relational weight domain.*

Definition 8. *A **weighted relation** on a set G, with weight domain $\mathcal{S} = (D, \oplus, \otimes, \overline{0}, \overline{1})$, is a function from $(G \times G)$ to D. The composition of two weighted relations R_1 and R_2 is defined as $(R_1; R_2)(g_1, g_3) = \oplus\{w_1 \otimes w_2 \mid \exists g_2 \in G : w_1 = R_1(g_1, g_2), w_2 = R_2(g_2, g_3)\}$. The union of the two weighted relations is defined as $(R_1 \cup R_2)(g_1, g_2) = R_1(g_1, g_2) \oplus R_2(g_1, g_2)$. The identity relation is the function that maps each pair (g, g) to $\overline{1}$ and others to $\overline{0}$. The reflexive transitive closure is defined in terms of these operations, as usual. If R is a weighted relation and $R(g_1, g_2) = z$, then we write $g_1 \xrightarrow{z} g_2 \in R$.*

Definition 9. *If $\mathcal{S}$ is a weight domain with set of weights D and G is a finite set, then the **relational weight domain** on $(G, \mathcal{S})$ is defined as $(2^{G \times G \rightarrow D}, \cup, ;, \emptyset, id)$: weights are weighted relations on G, combine is union, extend is weighted relational composition (";"), $\overline{0}$ is the empty relation, and $\overline{1}$ is the weighted identity relation on $(G, \mathcal{S})$.*

This weight domain can be encoded symbolically using techniques such as algebraic decision diagrams [11].

The weight domain of $\mathcal{E}_A$ will be a relational weight domain on $(G, \mathcal{S})$, where G encodes the state space of A, and $\mathcal{S}$ is the weight domain of $\mathcal{E}$. Intuitively, for a run ρ of $\mathcal{E}_A$, the valuation $val(\rho)$ in $\mathcal{E}_A$ is a weighted relation R such that if $q_1 \xrightarrow{z} q_2 \in R$, then (i) the valuation $val(\rho)$ in $\mathcal{E}$ must be equal to z, and (ii) starting from state q_1, A can make a transition to state q_2 on the nested word $post(\rho)$. We now introduce some notation needed to show how this is accomplished by the construction.

[3] $post[r](nw)$ is not always defined because of *max*; and thus neither is *post*. However, for a run of a PDS from the initial configuration, both will always be defined.

First, for an NWA $A = (Q, \Sigma, q_0, \delta, F)$, we define $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$. The relational weight domain of $\mathcal{E}_A$ is over the finite set $Q \times \Sigma_\epsilon$. The pairing of Q with Σ_ϵ is used below to properly model the return relation δ_r of A. We denote an element (q, σ) of this set by q^σ, but omit σ when $\sigma = \epsilon$.

Second, we define the restriction of δ_i to σ, denoted by $\delta_i^{|\sigma}$, to be the relation with $(q_1, q_2) \in \delta_i^{|\sigma}$ iff $(q_1, \sigma, q_2) \in \delta_i$. Note that by representing (q_1, q_2) as $(q_1^\epsilon, q_2^\epsilon)$, $\delta_i^{|\sigma}$ can be embedded into $(Q \times \Sigma_\epsilon) \times (Q \times \Sigma_\epsilon)$ using only states in which $q \in Q$ is paired with ϵ (i.e., q^ϵ). Henceforth, we abuse notation and use $\delta_i^{|\sigma}$ to mean the version that is embedded in $(Q \times \Sigma_\epsilon) \times (Q \times \Sigma_\epsilon)$. We define $\delta_c^{|\sigma}$ similarly. $\delta_i^{|\sigma}$ and $\delta_c^{|\sigma}$ will be the relational part of the weights that annotate step and push rules in $\mathcal{E}_A$. By restricting δ_i (δ_c) to σ, a run of $\mathcal{E}_A$ enforces that $\mathcal{E}$ and A are kept in lock step (see *Construction 1*).

Third, we define the function *expand*(σ), which takes as input a symbol $\sigma \in \Sigma$ and generates the relation $\{(q^\epsilon, q^\sigma) \mid q \in Q\}$. This is used to pass the return location to $\mathcal{E}_A$'s merging functions, which is needed for properly modeling the return relation δ_r of A.

Fourth, we define $\hat{\delta}$ so that $(q_r^\sigma, q_c, q) \in \hat{\delta}$ iff $(q_r, q_c, \sigma, q) \in \delta_r$. Notice that $\hat{\delta}$ combines the input symbol σ used in δ_r with the return state. This is used by $\mathcal{E}_A$'s merging functions to receive the return location passed via *expand*.

Construction 1. The combination of an EWPDS $\mathcal{E} = (\mathcal{P}, \mathcal{S}, f, g)$ and an NWA $A = (Q, \Sigma, q_0, \delta, F)$ is modeled by an EWPDS $\mathcal{E}_A$ that has the same underlying PDS as $\mathcal{E}$, but with a new weight domain and new assignments of weights and merging functions to rules: $\mathcal{E}_A = (\mathcal{P}_A, \mathcal{S}_A, f_A, g_A)$, where $\mathcal{P}_A = \mathcal{P}$, $\mathcal{S}_A = (D_A, \oplus_A, \otimes_A, \overline{0}_A, \overline{1}_A)$ is the relational weight domain on the set $Q \times \Sigma_\epsilon$ and weight domain $\mathcal{S}$, and f_A and g_A are defined as follows:

1. For step rule $r = \langle p, n_1 \rangle \hookrightarrow \langle p', n_2 \rangle \in \Delta$, $f_A(r) = \{q_1 \xrightarrow{f(r)} q_2 \mid (q_1, q_2) \in \delta_i^{|n_1}\}$.
2. For push rule $r = \langle p, n_c \rangle \hookrightarrow \langle p', e\ r_c \rangle \in \Delta$, $f_A(r) = \{q_1 \xrightarrow{f(r)} q_2 \mid (q_1, q_2) \in \delta_c^{|n_c}\}$ and

$$g_A(r)(w_c, w_x) =$$
$$\left\{ q_1 \xrightarrow{z} q_2 \mid \exists a, b : \begin{pmatrix} q_1 \xrightarrow{z_1} a \ \in w_c \\ \wedge\ a \xrightarrow{z_2} b \ \in (f_A(r) \otimes w_x) \\ \wedge\ \hat{\delta}(b, a, q_2) \end{pmatrix}, z = g(r)(z_1, z_2) \right\}$$

3. For pop rule $r = \langle p, x \rangle \hookrightarrow \langle p', \epsilon \rangle \in \Delta$, $f_A(r) = \{q \xrightarrow{f(r)} q^x \mid (q, q^x) \in$ *expand*$(x)\}$.

The properties of *Construction 1* are that (i) $\mathcal{E}_A$'s nested-word language is the intersection of those of $\mathcal{E}$ and A, and (ii) the behaviors of $\mathcal{E}_A$ (summarized by its MOVP values) are those of $\mathcal{E}$ restricted by A. Formally, these are captured by Thm. 1 and Cor. 1.

Theorem 1. *An NWA A combined with an EWPDS $\mathcal{E}$ results in an EWPDS $\mathcal{E}_A$ such that $L^{\varphi}(\mathcal{E}_A) = L(A, Q) \cap L(\mathcal{E})$, where for a run ρ of $\mathcal{E}_A$ with $z = val(\rho)$, $\varphi(z) = \exists q \in Q : q_0 \xrightarrow{y} q \in z$, and $y \neq \overline{0}$.*

Proof. See [12].

Corollary 1. *An NWA A combined with an EWPDS $\mathcal{E}$ results in an EWPDS $\mathcal{E}_A$ such that* $\text{MOVP}_{\mathcal{E}_A}(C) = \bigoplus\{val(\rho) \mid \rho \in Runs(\mathcal{E}, c), c \in C, post(\rho) \in L(A, Q)\}$.

Complexity of $\mathcal{E}_A$ versus $\mathcal{E}$. The complexity of computing MOVP on an EWPDS is proportional to the *height* of the weight domain, which is defined to be the length of the longest descending chain in the domain.[4] If H is the height of the weight domain of $\mathcal{E}$, then the height of the weight domain of $\mathcal{E}_A$ is $H|Q|^2$, where Q is the set of states of A. Because $\mathcal{E}$ and $\mathcal{E}_A$ have the same PDS, the complexity of computing MOVP on $\mathcal{E}_A$ only increases by a factor of $|Q|^2$.

5 Language Strength Reduction for the Empire Tool

Thm. 1 and Cor. 1 show that the EWPDS $\mathcal{E}_A$ created by *Construction 1* is able to model both $\mathcal{E}$ and A simultaneously (for nested words in their intersection). This capability allowed us to use language strength reduction to improve the Empire tool's performance. To make the discussion clear, we focus on EML process `P0` from Fig. 1. The first three steps are as follows:

1. $\mathcal{E}\langle\texttt{P0}\rangle$ is generated using the original Empire translation. Recall that the weight domain of $\mathcal{E}\langle\texttt{P0}\rangle$ is the prefix weight domain.
2. Let *Locks* be the set of locks of the EML program. For each lock $l \in Locks$, an NWA $A_l\langle\texttt{P0}\rangle$ is generated using the NWA template from §3.3. Define $A\langle\texttt{P0}\rangle$ to be $\bigcap_{l \in Locks} A_l\langle\texttt{P0}\rangle$. The state space Q of $A\langle\texttt{P0}\rangle$ is equal to $2^{|Locks|}$. That is, each $q \in Q$ represents a set of locks that are held. Note that in Fig. 1, there is only one lock l, and thus $A\langle\texttt{P0}\rangle = A_l\langle\texttt{P0}\rangle$, and $Q = \{\square, \boxtimes\}$.
3. $\mathcal{E}_A\langle\texttt{P0}\rangle$ is generated from $\mathcal{E}\langle\texttt{P0}\rangle$ and $A\langle\texttt{P0}\rangle$ using *Construction 1*. The NWA template from §3.3 is instantiated for $A\langle\texttt{P0}\rangle$, and thus $L(A\langle\texttt{P0}\rangle) = L(\mathcal{E}\langle\texttt{P0}\rangle)$. Hence, $\mathcal{E}_A\langle\texttt{P0}\rangle$ contains the same behaviors as $\mathcal{E}\langle\texttt{P0}\rangle$. Additionally, due of Thm. 1, $\mathcal{E}_A\langle\texttt{P0}\rangle$ is able to distinguish between OC and nOC lock acquisitions and releases in the same manner as $A\langle\texttt{P0}\rangle$.

From Fig. 2(a) to Fig. 2(b) The weight domain of $\mathcal{E}_A\langle\texttt{P0}\rangle$ is a relational weight domain over Q and the prefix weight domain of $\mathcal{E}\langle\texttt{P0}\rangle$. In $\mathcal{E}\langle\texttt{P0}\rangle$, the rule $r = \langle p, n_{12}\rangle \hookrightarrow \langle p, e_{\text{set}}\ x_{\text{testAndSet}}\rangle$ is annotated with the weight $\{\ (\ \}$. In $\mathcal{E}_A\langle\texttt{P0}\rangle$, r is annotated with the weight

[4] EWPDSs can also be used when the height is unbounded, provided there are no infinite descending chains. To simplify the discussion of complexity, we assume the height to be finite.

Table 2. For *Path 1* of $\mathcal{E}_A\langle \texttt{P0}\rangle$, a prefix bound of 7, and $\rho = [r_1, \ldots, r_{14}]$ from Fig. 4, cols. (a) and (b) present $val(\rho)(\square, \square)$ before and after distinguishing between OC and nOC lock acquisitions and releases, respectively. Col. (c) presents $val(\rho)(\square, \square)$ after removing all nOC lock acquisitions and releases from $\mathcal{E}_A\langle \texttt{P0}\rangle$. Note that for cols. (a) and (b), the valuation is an approximation, whereas col. (c) is able to describe *Path 1* exactly within the given prefix bound.

(a)	(b)	(c)
$[((R_v)(W_v\mathbf{\Sigma}^*$	$[(_o(_nR_v)_n(_nW_v\mathbf{\Sigma}^*$	$[(_oR_vW_v)_o]$

$$R = \{\square \xrightarrow{\{\ (\ \}} \boxtimes, \boxtimes \xrightarrow{\{\ (\ \}} \boxtimes\}$$

Observe that the state space of $A\langle \texttt{P0}\rangle$ is encoded in the weight, and that $R(\square, \boxtimes)$ denotes an OC lock acquisition, and $R(\boxtimes, \boxtimes)$ denotes an nOC lock acquisition. This is represented in R by annotating the two open-parenthesis symbols with the open and nested subscripts, respectively:

$$R = \{\square \xrightarrow{\{\ (_o\ \}} \boxtimes, \boxtimes \xrightarrow{\{\ (_n\ \}} \boxtimes\}$$

In other words, we perform the following transformation: For a weighted relation R, if $R(\square, \boxtimes) = \{\ (\ \}$, then $R(\square, \boxtimes) = \{\ (_o\ \}$; and if $R(\boxtimes, \boxtimes) = \{\ (\ \}$, then $R(\boxtimes, \boxtimes) = \{\ (_n\ \}$. Performing this transformation, and its dual for lock releases, on the weight of each rule and merging function induces a homomorphism, with respect to lock acquisitions and releases, from Fig. 2(a) to Fig. 2(b), on the language computed by MOVP($\mathcal{E}_A\langle \texttt{P0}\rangle$). This is illustrated by the weighted valuations for *Path 1* in Tab. 2, columns (a) and (b).

From Fig. 2(b) to Fig. 2(c). Once $\mathcal{E}_A\langle \texttt{P0}\rangle$ is able to distinguish between OC and nOC lock acquisitions and releases, we leverage *Observation 1* to remove all nOC lock acquisitions and releases. For a weighted relation R, if $R(\boxtimes, \boxtimes) = \{\ (_n\ \}$, then we set $R(\boxtimes, \boxtimes) = \overline{1}$. Performing this transformation, and its dual for lock releases, induces a homomorphism on the language from Fig. 2(b) to Fig. 2(c). This is exemplified by the weighted valuation of *Path 1* in Tab. 2, columns (b) and (c). Note that the valuation shown in column (c) is *not* an approximation like those in columns (a) and (b). This is because the string that describes *Path 1* is shorter after performing language strength reduction.

Removing nested parentheses that denote nOC acquisitions and releases guarantees that all lock acquisitions and releases modeled by $\mathcal{E}$ are OC. Thus, all EML locks can now be modeled in the CPDS by the trivial language shown in Fig. 2(c). Because the open and close-parenthesis symbols for nOC acquisitions and releases have been removed, a path in an EML process that uses reentrant locking can now be described by a shorter string. This can be seen in Tab. 2. In fact, there is now no cost to model a successive synchronized call, including recursive synchronized functions. Thus, in some cases, the CPDS model checker can find the same counterexample using a smaller bound k.

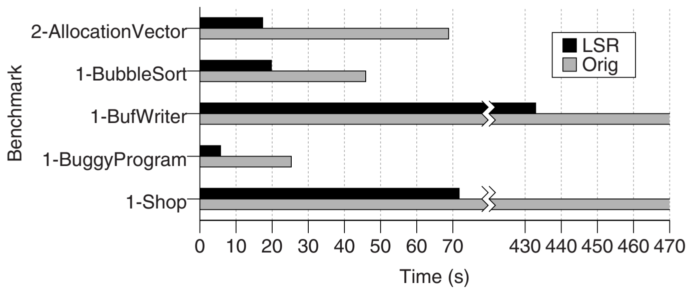


Fig. 5. Execution time for the CPDS model checker with the original encoding (Orig) and after language strength reduction (LSR)

6 Experiments

We implemented *Construction 1* and the transformations from §5 in the Empire tool. Five Java benchmark programs from the concurrency-testing benchmark by Eytani et al. [8] were analyzed. All experiments were run on a dual-core 3 GHz Pentium Xeon processor with 16 GB of memory. The machine ran a Windows XP Professional x64 Edition host OS, and an Ubuntu guest OS configured with the 32-bit Linux kernel 2.6.22. Ubuntu ran on top of VMware Server 1.0.4. A virtual machine was required because the CPDS model checker is only 32-bit Linux compatible.

Each benchmark was analyzed with the original encoding (Orig) and then again after applying language strength reduction (LSR). The analysis times are shown in Fig. 5. The Y-axis of Fig. 5 gives the benchmark names, with each name being preceded by the number of locks in the EML program (e.g., “BufWriter” uses 1 lock). For benchmark programs “AllocationVector”, “BubbleSort”, and “BuggyProgram”, the average speedup (geometric mean) is 3.4. In addition, analysis of the benchmark programs “BufWriter” and “Shop” exhausted all resources in the original version of Empire, whereas the analysis ran to completion after performing language strength reduction.

7 Related Work

Alur and Madhusudan [5,6] introduced the concept of an NWA. For program verification, they showed that a property specification and a program can be modeled by NWAs, and that verification can be solved by taking their intersection. Our work extends this result to property checking where the program is specified by an EWPDS and the property by an NWA. Because EWPDSs allow programs to be abstracted using more than just predicate-abstraction domains (i.e., abstract programs can be more than just Boolean programs), our work has broadened the class of program abstractions for which one can use an NWA as the property specification.

Chaudhuri and Alur [13] instrument a `C` program with an NWA that defines a property specification. This approach diffuses the NWA throughout the program proper. Our approach combines the NWA with an EWPDS, but keeps the NWA separated by modeling it using weights. This is beneficial for reporting error-paths back to a user when model checking a C program because the internals of the NWA are not exposed in the error-path. Additionally, by keeping the NWA separated in the weight domain, one can use symbolic encoding of weights [9] for handling the potentially exponential size of the NWA.

Kahlon et al. [14,15] analyze concurrent recursive programs that use nested locking, where nested locking means that all locks are released in the opposite order in which they are acquired. Their locks, however, are not reentrant and are not syntactically scoped. If one enforces syntactically scoped locks, then one can apply our techniques for language strength reduction to model a program with reentrant locks using only non-reentrant locks. This would produce a model to which their model-checking algorithm could be applied.

References

1. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL (2006)
2. Kidd, N., Reps, T., Dolby, J., Vaziri, M.: Static detection of atomic-set serializability violations. Technical Report TR-1623, Univ. of Wisconsin (October 2007)
3. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: POPL (2003)
4. Chaki, S., Clarke, E.M., Kidd, N., Reps, T.W., Touili, T.: Verifying concurrent message-passing C programs with recursive calls. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006 and ETAPS 2006. LNCS, vol. 3920, pp. 334–349. Springer, Heidelberg (2006)
5. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC (2004)
6. Alur, R., Madhusudan, P.: Adding nesting structure to words. In: H. Ibarra, O., Dang, Z. (eds.) DLT 2006. LNCS, vol. 4036, Springer, Heidelberg (2006)
7. Lal, A., Reps, T., Balakrishnan, G.: Extended weighted pushdown systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, Springer, Heidelberg (2005)
8. Eytani, Y., Havelund, K., Stoller, S.D., Ur, S.: Towards a framework and a benchmark for testing tools for multi-threaded programs. Conc. and Comp.: Prac. and Exp. 19(3) (2007)
9. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, TUM (2002)
10. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. In: SCP (2005)
11. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. In: CAD (1993)
12. Kidd, N., Lal, A., Reps, T.: Advanced queries for property checking. Technical Report TR-1621, Univ. of Wisconsin (October 2007)
13. Chaudhuri, S., Alur, R.: Instrumenting C programs with nested word monitors. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595. Springer, Heidelberg (2007)
14. Kahlon, V., Ivancic, F., Gupta, A.: Reasoning about threads communicating via locks. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, Springer, Heidelberg (2005)
15. Kahlon, V., Yang, Y., Sankaranarayan, S., Gupta, A.: Fast and accurate static data-race detection for concurrent programs. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, Springer, Heidelberg (2005)

Analysing All Polynomial Equations in $\mathbb{Z}_{2^w}$

Helmut Seidl, Andrea Flexeder, and Michael Petter

Technische Universität München,
Boltzmannstrasse 3, 85748 Garching, Germany
{seidl,flexeder,petter}@cs.tum.edu,
http://www2.cs.tum.edu/~{seidl,flexeder,petter}

Abstract. In this paper, we present methods for checking and *inferring all* valid polynomial relations in $\mathbb{Z}_{2^w}$. In contrast to the infinite field $\mathbb{Q}$, $\mathbb{Z}_{2^w}$ is finite and hence allows for finitely many polynomial functions only. In this paper we show, that checking the validity of a polynomial invariant over $\mathbb{Z}_{2^w}$ is, though decidable, only *PSPACE*-complete. Apart from the impracticable algorithm for the theoretical upper bound, we present a feasible algorithm for verifying polynomial invariants over $\mathbb{Z}_{2^w}$ which runs in polynomial time if the number of program variables is bounded by a constant. In this case, we also obtain a polynomial-time algorithm for inferring all polynomial relations. In general, our approach provides us with a feasible algorithm to infer all polynomial invariants up to a low degree.

1 Introduction

In reasoning about termination of programs, the crucial aspect is the knowledge about program invariants. Therefore, it is not surprising that the field of checking and finding of program invariants has been quite active, recently.

Many analyses interpret the values of variables regarding the field $\mathbb{Q}$. Modern computer architectures, on the other hand, provide arithmetic operations modulo suitable powers of 2. It is well-known that there are equalities valid modulo 2^w, which do not hold in general. The polynomial $2^{31}\mathbf{x}(\mathbf{x}+1)$, for example, constantly evaluates to 0 modulo 2^{32} but may show non-zero values over $\mathbb{Q}$. Accordingly, an analysis based on $\mathbb{Q}$ will systematically miss a whole class of potential program invariants.

```
1     int b = ?;
2     int c = 1 << 31, y = 0, x = 0;
3     while(y-b!= 0){
4       x = c*x*x + (c+1)*x + 1;
5       y = x*x + y;
6     }
```

Fig. 1. Computing the square power sum on 32bit machines

Example 1. As an example, consider the program from figure 1. This program repeatedly increases the value of program variable x in line 4 by 1 – if arithmetic is modulo 2^{32}. Therefore, the program powersum() computes a square sum. Thus, at program line 6 the polynomial invariant $2 \cdot \mathbf{x}^3 + 3 \cdot \mathbf{x}^2 + \mathbf{x} - 6 \cdot \mathbf{y} = 0$ holds modulo 2^{32} — but not over the field $\mathbb{Q}$.

© Springer-Verlag Berlin Heidelberg 2008

An exact analysis of the example program should take into account the structure of polynomials over the domain $\mathbb{Z}_{2^{32}}$: The right hand side in the assignment in line 4 of the example can be rewritten as $2^{31}\mathbf{x}(\mathbf{x}+1)+\mathbf{x}+1$ where the first summand $2^{31}\mathbf{x}(\mathbf{x}+1)$ is equivalent to the zero polynomial over $\mathbb{Z}_{2^{32}}$. Such polynomials are called *vanishing*. Singmaster [17] investigates the special structure of univariate vanishing polynomials over $\mathbb{Z}_m$ and provides necessary and sufficient conditions for a polynomial to vanish over $\mathbb{Z}_m$. Hungerbühler and Specker extend this result to multivariate polynomials and introduce a *canonical form* for polynomials in quotient rings [3]. Shekhar et.al. present an algorithm to compute this canonical representation over the quotient ring $\mathbb{Z}_{2^w}$ [16]. A minimal Gröbner base characterising all vanishing polynomials in arbitrary quotient rings is given by Wienand in [18]. In contrast to the infinite field $\mathbb{Q}$, the ring $\mathbb{Z}_{2^w}$ is finite. Therefore, there are just finitely many distinct k-ary polynomial functions. In fact, it will turn out that we can restrict ourselves to polynomials in k variables up to a total degree $1.5(w+k)$. Due to this upper bound on the total degrees of the polynomials of interest, the problem of checking or inferring of polynomials over $\mathbb{Z}_{2^w}$ becomes an analysis problem over finite domains only and therefore trivially is computable. Hence, the key issue is to provide tight upper complexity bounds as well as algorithms which also show decent behaviour on practical examples.

In this paper, we first consider the problem of checking whether a given polynomial relation is valid at a given program point. While being decidable over $\mathbb{Q}$, we show that this problem becomes *PSPACE*-complete over $\mathbb{Z}_{2^w}$. Furthermore, we present a practical algorithm for this problem which is based on effective precise weakest precondition computation. In case that the number of variables is bounded by a (small) constant, this algorithm even runs in polynomial time.

Secondly, we consider the problem of inferring all polynomial relations which are valid at a given program point. This problem, though not known to be computable in $\mathbb{Q}$, turns out to be computable in exponential time over $\mathbb{Z}_{2^w}$. Again, we present an algorithm for inferring all polynomial invariants of a given shape, whose runtime turns out to be polynomial given that the number of variables is bounded by a constant. Both algorithms have been implemented, and we report on preliminary experiments.

Related Work

The pioneer in the area of finding polynomial relations was Karr [4] who inferred the validity of polynomial relations of degree at most 1 (i.e., affine relations) over programs using affine assignments and tests only. An algorithm for checking validity of polynomial relations over programs using polynomial assignments is provided by Müller-Olm and Seidl [7] and was extended later to deal with disequality guards as well [9]. Their approach is based on effective weakest precondition computations where conjunctions of polynomial relations are described by *polynomial ideals*. Termination of a fixpoint computation in $\mathbb{Q}$ thus is guaranteed by Hilbert's base theorem. In [9], the authors also observe that their method for checking the validity of polynomial relations can be used to construct an algorithm for inferring all polynomial invariants up to a fixed degree. In [13,14] Rodriguez-Carbonell et al. pick up the idea of describing invariants by polynomial ideals and propose a *forward* propagating analysis, based on a constraint system over these ideals. As infinite *descending* chains of polynomial ideals cannot be avoided in $\mathbb{Q}$ when merging execution paths [8, Example 1], they provide special cases or widening techniques to infer polynomial identities. Sankaranarayanan et al. also investigate polynomial invariants[15]. They propose to use *polynomial templates* to capture the effect of assignments in their analysis. These templates describe parametric polynomial properties. When determining the generic parameters via Gröbner bases, certain inductive invariants can be inferred. In contrast to the former approaches, Colon [2] provides an *interprocedural forward analysis* for polynomial programs. This analysis is based on ideals of polynomial *transition invariants*. In order to deal with infinite descending chains, Colon abstracts ideals with *pseudo-ideals*, which essentially are vector spaces of polynomials up to a given degree. The application of weakest precondition computations to interprocedural analysis of polynomial relations over $\mathbb{Q}$ is discussed in [8]. An exact (even interprocedural) analysis of affine relations for programs using affine assignments over the domain $\mathbb{Z}_m$ is provided in [10,11].

This paper is organised as follows. In Section 2 we specify the concrete semantics for the program class that is inspected by our analysis by means of control flow graphs. Section 3 gives a detailed description of the characteristics of polynomials in $\mathbb{Z}_{2^w}$. In Section 4, we first provide the complexity class for the general case of verifying polynomial invariants in $\mathbb{Z}_{2^w}$. We then specify our abstraction for the concrete semantics with the help of polynomial ideals. In Section 5 we present our specific concrete representation of polynomial ideals in $\mathbb{Z}_{2^w}$. We show how they contribute in the case of constantly many program variables to a better runtime complexity than the theoretical worst case in Section 4. We then illustrate in Section 6, how to extend this procedure to infer valid invariants up to a fixed degree and thus for inferring all valid relations. Section 7 finally summarises our results.

2 Fixpoint Semantics

In this section we introduce the programs to be analysed together with the concrete semantics our polynomial analysis is based on. Basically, we emanate from the same concrete semantics as in [9].

The vector of variables $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_k)$ from the set of program variables $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_k\}$ can take values in the ring $\mathbb{Z}_{2^w}$. A program state, which assigns values to variables, can be modelled by a k-dimensional vector $x = (x_1, \ldots, x_k) \in \mathbb{Z}_{2^w}^k$, where x_i is the value assigned to variable $\mathbf{x}_i$.

We assume that the basic statements in the considered program class are either *polynomial assignments* of the form $\mathbf{x}_j := p$ or *non-deterministic assignments* of the form $\mathbf{x}_j :=?$ where $\mathbf{x}_j \in \mathbf{X}$ or *polynomial disequality guards* of the form $p \neq 0$ where $p \in \mathbb{Z}_{2^w}[\mathbf{X}]$. Recalling, that finding polynomial invariants in presence of equality guards turns out to be indecidable, we keep to non-deterministic branching instead. Non-deterministic assignments $\mathbf{x}_j :=?$ represent a safe abstraction of statements our analysis cannot handle precisely, e.g. non-polynomial expressions or user input.

Let Lab denote the set of basic statements and polynomial disequality guards. A *polynomial program* is given by a non-deterministic *control flow graph*, consisting of:

- *program points* N,
- a set of edges $E \subseteq N \times N$,
- a mapping $A : E \to \mathsf{Lab}$ from edges to statements or polynomial disequality guards
- a special *start point* $\mathsf{st} \in N$.

The program executions reaching a given program point are characterised by a constraint system, for which our analysis provides a precise abstract interpretation. A program execution r (also called *run*) is a finite sequence $r \equiv r_1; \ldots ; r_m$ where each r_i is a basic statement or disequality guard. Runs denotes the set of runs, which can be characterised as the smallest solution of a system of subset constraints on run sets $\mathbf{R}$, reaching the target program point t.

$$\begin{array}{l}[\mathrm{R1}]\ \mathbf{R}(t) \supseteq \{\varepsilon\} \\ [\mathrm{R2}]\ \mathbf{R}(u) \supseteq f_e(\mathbf{R}(v))\,,\ \text{if } e = (u, v) \in E\end{array}$$

Constraint [R1] expresses, that the set of runs reaching program point t when starting from t contains the empty run, denoted by "ε". By [R2], a run starting from u is obtained by considering an outgoing edge $e = (u, v)$ and concatenating a run corresponding to e with a run starting from v, where $f_e(R) = \{r; t \mid r \in \mathbf{R}(e) \wedge t \in R\}$. If edge e is annotated by $A(e) \equiv p \neq 0$ or $A(e) \equiv \mathbf{x}_j := p$, it gives rise to a single execution: $\mathbf{R}(e) = \{A(e)\}$. The effect of an edge e annotated by $\mathbf{x}_j :=?$ is captured by collecting *all constant* assignments:

$$\mathbf{R}(e) = \{\mathbf{x}_j := c \mid c \in \mathbb{Z}_{2^w}\}$$

Each run induces a *partial* transformation of the underlying program state $x \in \mathbb{Z}_{2^w}^k$. In the case of a disequality guard $p \neq 0$ this results in a partial identity function:

$$\mathsf{dom}([\![p \neq 0]\!]) \;=\; \{x \in \mathbb{Z}_{2^w}^k \mid p(x) \neq 0\}$$

A polynomial assignment $\mathbf{x}_j := p$ causes the transformation with $\mathsf{dom}([\![\mathbf{x}_j := p]\!]) = \mathbb{Z}_{2^w}^k$ and

$$[\![\mathbf{x}_j := p]\!]\, x = (x_1, \ldots , x_{j-1}, p(x), x_{j+1}, \ldots , x_k)$$

Extending these definitions to runs, we obtain: $[\![\varepsilon]\!] = \mathsf{Id}$, where Id is the identity function and $[\![r; r_{rest}]\!] = [\![r_{rest}]\!] \circ [\![r]\!]$ where "$\circ$" denotes composition of partial functions. The partial transformation $f = [\![r]\!]$ induced by a run r can always be represented by polynomials $q_0, \ldots , q_k \in \mathbb{Z}_{2^w}[\mathbf{X}]$ such that $\mathsf{dom}(f) = \{x \in \mathbb{Z}_{2^w}^k \mid q_0(x) \neq 0\}$ and $f(x) = (q_1(x), \ldots , q_k(x))$ for every $x \in \mathsf{dom}(f)$. For the identity transformation induced by the empty path ε the polynomials $1, \mathbf{x}_1, \ldots , \mathbf{x}_k$ would hold. Transformations induced by polynomial assignments or guards can thus be represented in this manner and are closed under composition, similarly to [9].

3 The Ring of Polynomials in $\mathbb{Z}_{2^w}$

In order to develop an analysis inferring all polynomial relations modulo $m = 2^w$, we fix a bit width $w \geq 2$ of our numbers in the following. For $\mathbf{X} = \{\mathbf{x}_1, \ldots , \mathbf{x}_k\}$, let $\mathbb{Z}_{2^w}[\mathbf{X}]$ denote the ring of all polynomials with coefficients in $\mathbb{Z}_{2^w}$. Each polynomial p can be written as a sum of terms, i.e., have the form $p = \sum_\delta c_\delta \cdot \mathbf{x}_1^{\delta_1} \ldots \mathbf{x}_k^{\delta_k}$, with its degree $\delta \in \mathbb{N}^k, c \in \mathbb{Z}_{2^w}$ and $\mathbf{x} \in \mathbf{X}$. We call $\mathbf{x}_1^{\delta_1} \dots \mathbf{x}_k^{\delta_k}$ a monomial of total degree $\sum \delta_i$ and c_δ its coefficient. We agree on the terms for each polynomial to be sorted lexicographically on the string of degrees in the variables from $\mathbf{X}$. Then each polynomial p has a head term (respectively head coefficient or head monomial), that leads the trailing terms.

Recall that the coefficient ring $\mathbb{Z}_{2^w}$ is not a field. More precisely, only all odd elements are invertible while every even element is a *zero divisor*. Thus, e.g., $2 \cdot 2^{w-1} \equiv 0$ in $\mathbb{Z}_{2^w}$. Useful facts about this ring can be found in [10] or basic text books on commutative ring theory, as [5].

Similarly to the case of fields [10], the set of polynomials $p \in \mathbb{Z}_{2^w}[\mathbf{X}]$ which evaluate to 0 for a given subset $X \subseteq \mathbb{Z}_{2^w}^k$, is closed under addition and multiplication with arbitrary polynomials. A non-empty subset I of a ring with this property is also called an *ideal*. Thus, our program analysis maintains for every program point an ideal of polynomials.

Recall that the ring $\mathbb{Z}_{2^w}$ is a *principal ideal* ring meaning that every ideal $I \subseteq \mathbb{Z}_{2^w}$ can be represented as the set $I = \{z \cdot a \mid z \in \mathbb{Z}_{2^w}\}$ of all multiples of a single ring element a. Thus by Hilbert's basis theorem, every ideal $I \subseteq \mathbb{Z}_{2^w}[\mathbf{X}]$ can be represented as the set of all linear combinations of a finite set $G = \{g_1, \dots, g_n\} \subseteq \mathbb{Z}_{2^w}[\mathbf{X}]$, i.e., $I = \{p_1 g_1 + \dots + p_n g_n \mid p_i \in \mathbb{Z}_{2^w}[\mathbf{X}]\}$. In this case, we also refer to G as the set of generators of I and denote this by $I = \langle G \rangle$.

Assume $p, p' \in G$ are polynomials which share the same monomial t in their head-term, i.e., are of the form: $p = a \cdot 2^e \cdot s \cdot t + p_{rest}$ and $p' = a' \cdot 2^{e'} \cdot t + p'_{rest}$ with $e \geq e'$, some monomial s and odd $a, a' \in \mathbb{Z}_{2^w}$. In this case, we say that p is *reducible* by p'. More generally, we call p *reducible* by a set R of polynomials if p is reducible w.r.t. some $p' \in R$. If p is reducible by the polynomial p', p can be reduced to the polynomial $q = a' \cdot p - a \cdot 2^{e-e'} \cdot s \cdot p'$. If $q = 0$, p is a multiple of p' and thus redundant in every set G of generators containing p', i.e., $\langle G \backslash \{p\} \rangle = \langle G \rangle$. If $q \neq 0$, we can replace the polynomial p in G with the (simpler) polynomial q, i.e., the set G generates the same ideal as the set $G' = (G \backslash \{p\}) \cup \{q\}$.

Starting from a set G of generators, we can successively apply reduction to eventually arrive at a *reduced* set $\bar{G}$ generating the same ideal as G. Here, we call the set $\bar{G}$ reduced iff no polynomial $p \in \bar{G}$ is reducible w.r.t. $\bar{G} \backslash \{p\}$.

Lemma 1. *Assume that $p \in \mathbb{Z}_{2^w}[\mathbf{X}]$ and $G \subseteq \mathbb{Z}_{2^w}[\mathbf{X}]$ is a finite reduced set of polynomials. Then a reduced set $\bar{G} \subseteq \mathbb{Z}_{2^w}[\mathbf{X}]$ can be constructed with $\langle \{p\} \cup G \rangle = \langle \bar{G} \rangle$. The algorithm runs in time $\mathcal{O}(k \cdot r^2)$ if r is the number of different exponents of monomials occurring during reduction, and k the number of program variables.*

Proof. A single reduction of a polynomial by a set of reduced polynomials is carried out by as many subtractions (each of cost k) as a polynomial has monomials. This number is bounded by the number of different exponents r occuring during the reduction. Adding p to G is carried out by reducing potentially $|G| \leq r$ many polynomials. $\square$

Here, the total degree of a monomial $\mathbf{x}_1^{r_1} \dots \mathbf{x}_k^{r_k}$ is $d = r_1 + \dots + r_k$, and the total degree of a polynomial is the total degree of its head monomial. Thus, the number r of possibly occurring different exponents of monomials is bounded by:

Proposition 1. *The number of different exponents of monomials is given by*

$$r \leq \binom{d+k}{k} \leq min(\ (d+1)^k\ ,\ (k+1)^d\ )$$

Note that the upper complexity bound is a crude worst-case estimation only. The practical run-time might be much smaller if the occurring polynomials are short, i.e., contain only few monomials.

Now assume that the ideal I is generated from a set G of generators and p is a polynomial. If p can be reduced (perhaps in several steps) to the 0 polynomial by means of the polynomials in G, then $p \in I$. The reverse, however, is only true for particularly saturated sets of generators such as *Gröbner bases* [1].

In the case of polynomials over a field, the constant zero polynomial is the only polynomial which evaluates to 0 for all vectors $x \in \mathbb{Z}_{2^w}^k$. This is no longer the case for the polynomial ring $\mathbb{Z}_{2^w}[\mathbf{X}]$. Let $I_v \subseteq \mathbb{Z}_{2^w}[\mathbf{X}]$ denote the ideal of all polynomials p with $p(x) = 0$ for all $x \in \mathbb{Z}_{2^w}^k$. The elements of I_v are also called *vanishing* polynomials. Only recently, a precise characterisation of the ideal I_v has been provided by Hungerbühler and Specker [3], which is recalled briefly, here. The first observation is that whenever 2^e divides $r! = r(r-1)\dots 1$, then 2^e also divides $(x+r-1)\cdot\ldots\cdot(x+1)\cdot x$ for all x. Let $\nu_2(y)$ denote the maximal exponent e such that 2^e divides y. Thus, e.g., $\nu_2(1!) = 0$, $\nu_2(2!) = \nu_2(3!) = 1$ and $\nu_2(2^s!) = 2^s - 1$ for all $s \geq 1$. In particular, $\nu_2(r!) \geq w$ for $r \geq w + \log(w) + 1$. Since $1.5w + 1 \geq w + \log(w) + 1$, this implies that $\nu_2(r!) \geq \frac{2}{3}r - 1$.

Now consider the polynomial $p_r(\mathbf{x}_i) = \mathbf{x}_i \cdot (\mathbf{x}_i + 1) \cdot \ldots \cdot (\mathbf{x}_i + r - 1)$. Then $2^{\nu_2(r!)}$ divides the value $p_r(z)$ for every z. Thus we obtain the following family $G(k, w)$ of vanishing polynomials:

$$2^a \cdot p_{r_1}(\mathbf{x}_1) \cdot \ldots \cdot p_{r_k}(\mathbf{x}_k)$$

where $a \geq 0$, and $a + \nu_2(r_1!) + \ldots + \nu_2(r_k!) \geq w$.

Example 2. Take $\mathbb{Z}_{2^2}$ as domain. Then $p = \mathbf{x}^4 + 2\mathbf{x}^3 + 3\mathbf{x}^2 + 2\mathbf{x} = \mathbf{x}(\mathbf{x}+1)(\mathbf{x}+2)(\mathbf{x}+3) = p_4(\mathbf{x})$. Since $\nu_2(4!) = 3 \geq 2$, $p(\mathbf{x})$ is a vanishing polynomial. $\square$

Note that Wienand [18] proves that the set $G(k, w)$ is not only contained in I_v but that I_v is in fact generated by $G(k, w)$.

Two polynomials $p, p' \in \mathbb{Z}_{2^w}[\mathbf{X}]$ are *semantically* equivalent if they define the same function $\mathbb{Z}_{2^w}^k \to \mathbb{Z}_{2^w}$, i.e., if $p - p' \in I_v$. We observe that for every polynomial in $\mathbb{Z}_{2^w}[\mathbf{X}]$, we can effectively find a semantically equivalent polynomial of small total degree. We have:

Lemma 2. *Every polynomial* $p \in \mathbb{Z}_{2^w}[\mathbf{X}]$ *in* k *variables is semantically equivalent to a polynomial* $p' \in \mathbb{Z}_{2^w}[\mathbf{X}]$ *of degree less than* $1.5(w + k)$.

Proof. Let p' denote a polynomial of minimal total degree r and minimal number of monomials of total degree r which is semantically equivalent to p. Assume for a contradiction that $r \geq 1.5(w + k)$ and t is a monomial in p' of maximal total degree.

Then t can be written as $t = a \cdot \mathbf{x}_1^{r_1} \ldots \mathbf{x}_k^{r_k}$ where w.l.o.g. all $r_i \geq 1$. Then $\nu_2(r_j!) \geq \frac{2}{3}r_j - 1$ for all j. Consequently, the sum of these values is at least

$$\sum_j (\frac{2}{3}r_j - 1) = \frac{2}{3}\sum_j (r_j - 1.5) = \frac{2}{3}(1.5(w+k) - 1.5k) = w$$

Therefore, the polynomial $q = p_{r_1}(\mathbf{x}_1) \cdot \ldots \cdot p_{r_k}(\mathbf{x}_k)$ is vanishing. Note that the polynomial q has exactly one monomial of maximal total degree. We conclude that $p'' = p' - a \cdot q$ is a polynomial which is still semantically equivalent to p. Moreover the total degree of p'' is not larger than the total degree of p' and if the respective total degrees are equal, then p'' has less monomials of maximal total degree – thus contradicting our assumption. □

Example 3. Consider the polynomials $p = \mathbf{x}^4 + 3\mathbf{x}$ and $p' = 2\mathbf{x}^3 + \mathbf{x}^2 + \mathbf{x}$ over $\mathbb{Z}_{2^2}$. Subtracting p' from p results in $q = p - p' = \mathbf{x}^4 + 2\mathbf{x}^3 + 3\mathbf{x}^2 + 2\mathbf{x} \in I_v$. Thus, p and p' are equivalent. □

In [16], Shekhar et al. prove that a polynomial p is vanishing iff p can be reduced to 0 by means of the polynomials in $G(k, w)$. In the worst case, this takes $\mathcal{O}((d+1)^k)$ reduction steps if d is the total degree of p. As each of these reductions involves $\mathcal{O}((d+1)^k)$ many different monomials in the worst case, checking a polynomial for vanishing by means of reduction costs $\mathcal{O}((d+1)^{2k})$. Here, we sketch an alternative method. It consists in evaluating the polynomial for a finite set of selected arguments. The latter technique is based on the following observation.

Lemma 3. *A polynomial $p \in \mathbb{Z}_{2^w}[\{\mathbf{x}\}]$ of degree d is semantically equivalent to the zero polynomial, i.e., $\forall x \in \mathbb{Z}_{2^w}.p(x) = 0$ iff $p(h) = 0$ for $h = 0, 1, \ldots, d$.*

Proof. "$\Rightarrow$" is trivial.
"$\Leftarrow$" by induction on degree d:
case $d = 0$*:* If the degree is zero, the polynomial is just described by a constant function $p(\mathbf{x}) \equiv c$. This polynomial is only zero for any h, if the constant value c is zero. Therefore $p \equiv 0$ and the assertion follows.
case $d > 0$*:* Let $p(h) = 0$ for $h = 0, \ldots, d$. We consider the polynomial q of degree $d - 1$ with $q(\mathbf{x}) = p(\mathbf{x}+1) - p(\mathbf{x})$ that has $q(h') = 0$ at least for $h' = 0, \ldots, d-1$. By induction hypothesis, $q(x) = 0$ for all $x \in \mathbb{Z}_{2^w}$. Thus, $p(\mathbf{x})$ and $p(\mathbf{x}+1)$ are semantically equivalent. Since $p(0) = 0$, then also $p(1) = 0$ and thus, by induction, $p(x) = 0$ for all $x \in \mathbb{Z}_{2^w}$, and the assertion follows. □

Lemma 3 shows that an arbitrary polynomial p is vanishing iff it vanishes for suitably many argument vectors. Substituting in a polynomial p of degree d all k different variables by $d + 1$ values each indicates that $p \notin I_v$, if it does not evaluate to zero each time. Otherwise $p \in I_v$. As evaluating p can be done in $|p|$, we conclude:

Corollary 1. *Assume $p \in \mathbb{Z}_{2^w}[\mathbf{X}]$ is a polynomial where each variable has a maximal degree in p bounded by d. Then $p \in I_v$ can be tested in time $\mathcal{O}((d+1)^k \cdot |p|)$.* □

4 Verifying Polynomial Relations in $\mathbb{Z}_{2^w}$

Similarly to [9] we denote a *polynomial relation* over the vector space $\mathbb{Z}_{2^w}^k$ as an equation $p = 0$ for some $p \in \mathbb{Z}_{2^w}[\mathbf{X}]$, which is representable by p alone. The vector $y \in \mathbb{Z}_{2^w}^k$ *satisfies* the polynomial relation p iff $p(y) = 0$. The polynomial relation p is valid at a program point v iff p is satisfied by $[\![r]\!]x$ for every run r of the program from program start st to v and every vector $x \in \mathbb{Z}_{2^w}^k$. In [6], Rüthing and Müller-Olm prove that deciding whether a polynomial relation over $\mathbb{Q}$ is valid at a program point v of a polynomial program is at least *PSPACE*-hard. Their lower-bound construction is based on a reduction of the *language universality problem* of non-deterministic finite automata and uses only the values 0 and 1. Therefore, literally the same construction also shows that validity of a polynomial relation over $\mathbb{Z}_{2^w}$ is also *PSPACE*-hard. Regarding an upper bound, we construct a Turing machine which non-deterministically computes a counterexample for the validity of a polynomial relation p. This counterexample can be found by simulating the original program on vectors over $\mathbb{Z}_{2^w}$ representing the program state. The representation of such a program state can be done in polynomial space in $\mathbb{Z}_{2^w}$. The Turing machine accepts if it reaches program point v with a state $x \in \mathbb{Z}_{2^w}^k$ which does not satisfy p. Thus, we have a *PSPACE*-algorithm for dis-proving the validity of polynomial relations. Since the complexity class *PSPACE* is closed under complementation, we obtain:

Theorem 1. *Checking validity of polynomial invariants over $\mathbb{Z}_{2^w}$ is PSPACE -complete.* □

This is bad news for a general algorithm for the verification of polynomial invariants over $\mathbb{Z}_{2^w}$. The theoretical algorithm providing the upper bound in theorem 1 is not suitable for practical application. Therefore, we subsequently present an algorithm which has reasonable runtime behaviour at least for meaningful examples. In particular, it has polynomial complexity — given that the number of program variables is bounded by a constant.

This algorithm is based on the effective computation of weakest preconditions. Following [9], we characterise the weakest precondition of the validity of a relation p_t at program point t by means of a constraint system on ideals of polynomials.

In order to construct this constraint system, we rely on the weakest precondition transformers $[\![s]\!]^\top$ for tests, single assignments, or non-deterministic assignments:

$$\begin{aligned} [\![p \neq 0]\!]^\top q &= \{p \cdot q\} \\ [\![\mathbf{x}_j := p]\!]^\top q &= \{q[p/\mathbf{x}_j]\} \\ [\![\mathbf{x}_j :=?]\!]^\top q &= \{q[h/\mathbf{x}_j] \mid h = 0, \ldots, d\} \end{aligned}$$

where d is the maximal degree of $\mathbf{x}_j$ in q. The transformers for assignments and disequality tests are the same which, e.g., have been used in [9]. Only for non-deterministic assignments $\mathbf{x}_j :=?$, extra considerations are necessary. The treatment of non-deterministic assignments in [9] for $\mathbb{Q}$ consists in collecting all coefficient polynomials p_i not containing $\mathbf{x}_j$ in the sum $q = \sum_{i \geq 0} p_i \cdot \mathbf{x}_j^i$. This idea does no longer work over $\mathbb{Z}_{2^w}$. Consider, e.g., $p = 2^{31}\mathbf{x}_1^2\mathbf{x}_2 + 2^{31}\mathbf{x}_1\mathbf{x}_2$. Equating every $\mathbf{x}_1$-coefficient with zero would lead to the polynomial $2^{31}\mathbf{x}_2 = 0$ — which is not the weakest precondition, as

$p = 0$ is trivially valid. The correctness of the new definition of $[\![\mathbf{x}_j :=?]\!]^\top$ for $\mathbb{Z}_{2^w}$ on the other hand, follows from lemma 3.

The weakest precondition transformers $[\![s]\!]^\top$ for polynomials can be extended to transformers of ideals. Assume that the ideal I is given through the set G of generators. Then:

$$[\![s]\!]^\top I = \langle \bigcup \{[\![s]\!]^\top g \mid g \in G\}\rangle$$

Note that $[\![s]\!]^\top q$ is vanishing whenever q is already vanishing. Therefore,

$$[\![s]\!]^\top (I_v) \subseteq I_v$$

for all s. Using the extended transformers, we put up the constraint system $\mathbf{R}^\sharp_{p_t}$ to represent the precondition for the validity of a polynomial p_t at program point t:

$$\begin{array}{ll}[\text{R1}]^\sharp\ \mathbf{R}^\sharp_{p_t}(t) \supseteq \langle\{p_\text{t}\}\rangle & \\ [\text{R2}]^\sharp\ \mathbf{R}^\sharp_{p_t}(u) \supseteq [\![s]\!]^\top(\mathbf{R}^\sharp_{p_t}(v))\,, & \text{if } e = (u,v) \in E \wedge A(e) \equiv s\end{array}$$

For all program points, we may safely assume that all vanishing polynomials are valid. Therefore, we may consider the given constraint system over ideals I subsuming I_v, i.e., with $I_v \subseteq I$. This implies that we only consider ideals I where $p \in I$ whenever $p' \in I$ for every polynomial p' which is semantically equivalent to p. Note that the set of ideals subsuming I_v (ordered by the subset relation '$\subseteq$") forms a complete lattice. Since all transformers $[\![s]\!]^\top$ are monotonic, this system has a unique least solution. Since all transformers $[\![s]\!]^\top$ transform I_v into (subsets of) I_v and distribute over sums of ideals, the least solution of the constraint system precisely characterises the weakest preconditions for the validity of p_t at program point t in a similar way as in [9]. We have:

Lemma 4. *Assume that $\mathbf{R}^\sharp_{p_t}(u)$, with u a program point, denotes the least solution of the constraint system $\mathbf{R}^\sharp_{p_t}$. Then the polynomial relation $p_\text{t} \in \mathbb{Z}_{2^w}[\mathbf{X}]$ is valid at the target node t iff $\mathbf{R}^\sharp_{p_t}(\mathsf{st}) \subseteq I_v$.* □

5 Computing with Ideals over $\mathbb{Z}_{2^w}[\mathbf{X}]$

In order to check the validity of the polynomial relation p_t at program point t, we must find succinct representations for the ideals occurring during fixpoint iteration which allow us first, to decide when the fixpoint computation can be terminated and secondly, to decide whether the ideal for the program start consists of vanishing polynomials only.

The basic idea consists in representing ideals through finite sets G of generators. In order to keep the set G small, we explicitly collect only polynomials *not* in I_v. Thus, G represents the ideal $\langle G\rangle_v = \langle G\rangle \oplus I_v = \{g + g_0 \mid g \in \langle G\rangle, g_0 \in I_v\}$.

Keeping the representation of vanishing polynomials implicit is crucial, since the number of necessary vanishing polynomials in $G(k,w)$ is exponential in k. By lemma 2, only polynomials up to degree $1.5(w+k)$ need to be chosen. By successively applying reduction, we may assume that G is reduced and consists of polynomials which cannot be (further) reduced by polynomials in $G(k,w)$ only. Let us call such sets of generators *normal-reduced*. Then by the characterisation of [16], $\langle G\rangle_v \subseteq I_v$ iff $G = \emptyset$.

Example 4. Consider the polynomials $p = \mathbf{x}^5 + \mathbf{x}^4 + 2\mathbf{x}^2 + 3\mathbf{x}$ and $\bar{p} = 6\mathbf{x}^5 + 7\mathbf{x}^3 + 2\mathbf{x}$ over $\mathbb{Z}_{2^3}$. In order to build a normal-reduced set of generators $G, \langle G\rangle_v = \langle\{p, \bar{p}\}\rangle_v$, we begin with a first round, reducing p and $\bar{p}$ with the vanishing polynomial $p_v = \mathbf{x}^4 + 2\mathbf{x}^3 + 3\mathbf{x}^2 + 2\mathbf{x}$: $p' = p + (7\mathbf{x}+1)p_v = 7\mathbf{x}^3 + 3\mathbf{x}^2 + 5\mathbf{x}$ and $\bar{p}' = \bar{p} + (2\mathbf{x}+4)p_v = 5\mathbf{x}^3 + 2\mathbf{x}$. Next, p' can be reduced by $\bar{p}'$, leading to $p'' = p' + 5\bar{p}' = 3x^2 + 7x$. $\bar{p}'$ and p'' are nonreducible with respect to each other and I_v. Then $\langle\{, p'', \bar{p}'\}\rangle_v = \langle\{p, \bar{p}\}\rangle_v$ where the set $\{p'', \bar{p}'\}$ is normal-reduced. □

Concerning the computation of the fixpoint for $\mathbf{R}^\sharp_{p_t}$, consider an edge $e = (u, v)$ in the control-flow graph of the program labelled with $s = A(e)$. Each time when a new polynomial p is added to the ideal $\mathbf{R}^\sharp_{p_t}(v)$ associated with program point v which is not known to be contained in $\mathbf{R}^\sharp_{p_t}(v)$, all polynomials in $[\![s]\!]^\top p$ must be added to the ideal $\mathbf{R}^\sharp_{p_t}(u)$ at program u. The key issue for detecting termination of the fixpoint algorithm therefore is to check whether a polynomial p is contained in the ideal $\mathbf{R}^\sharp_{p_t}(u)$. Assume that the ideal $\mathbf{R}^\sharp_{p_t}(u)$ is represented by the normal-reduced set G of generators. Clearly, the polynomial p is contained in $\langle G\rangle_v = \mathbf{R}^\sharp_{p_t}(u)$ whenever p can be reduced by $G \cup G(k, w)$ to the 0 polynomial. The reverse, however, need not necessarily hold.

Exact ideal membership based on *Gröbner bases* requires to extend the set G with *S-polynomials* [1]. However, for generating all S-polynomials, virtually all pairs of generators must be taken into account. This applies also to the vanishing polynomials. The number of vanishing polynomials in $G(k, w)$ of degree $\mathcal{O}(w + k)$, however, is still exponential in k and also may comprise polynomials with many monomials. This implies that any algorithm based on exhaustive generation of S-polynomials cannot provide decent mean- or best case complexity at least in some useful cases. Therefore, we have abandoned the generation of S-polynomials altogether, and hence also exact testing of ideal membership.

Instead of ideals themselves, we therefore work with the complete lattice $\mathbb{D}$ of normal-reduced subsets of polynomials in $\mathbb{Z}_{2^w}[\mathbf{X}]$. The ordering on the lattice $\mathbb{D}$ is defined by $G_1 \sqsubseteq G_2$ iff every element $g \in G_1$ can be reduced to 0 w.r.t. $G_2 \cup G(k, w)$. The least element w.r.t. this ordering is $\emptyset$. Thus by definition, $G_1 \sqsubseteq G_2$ implies $\langle G_1\rangle_v \subseteq \langle G_2\rangle_v$, and $\langle G\rangle_v = I_v$ iff $G = \emptyset$. In order to guarantee the termination of the modified fixpoint computation, we rely on the following observation:

Lemma 5. *Consider a strictly increasing chain:*

$$\emptyset \sqsubset G_1 \sqsubset \ldots \sqsubset G_h$$

of normal-reduced generator systems over $\mathbb{Z}_{2^w}[\mathbf{X}]$. *Then the maximal length* h *of this chain is bounded by* $w \cdot r$ *with* r *as the number of head monomials occurring in any* G_i.

Proof. For each G_i consider the set H_i, which denotes the set of terms $t = 2^s \cdot \mathbf{x}_1^{r_1} \ldots \mathbf{x}_k^{r_k}$ for which $a \cdot t$ (a invertible) is the head term of a polynomial in G_i. Then for every i, H_i contains a term t which has not yet occurred in any $H_j, j < i$. The value h thus is bounded by the cardinality of $H_1 \cup \ldots \cup H_h$, which is bounded by $w \cdot r$. □

In case that we are given an upper bound d for the total degree of polynomials in lemma 5, then by prop. 1, the height h is bounded by $h \leq w \cdot r \leq w \cdot (d+1)^k$ and thus is exponential in k only.

The representation of ideals through normal-reduced sets of generators allows us to compute normal-reduced sets of generators for the least solution of the constraint system $\mathbf{R}^\sharp_{p_t}$. We obtain:

Theorem 2. *Checking the validity of a polynomial invariant in a polynomial program with N nodes and k variables over $\mathbb{Z}_{2^w}$ can be performed in time $\mathcal{O}(N \cdot k \cdot w^2 \cdot r^3)$ where r is the number of monomials occurring during fixpoint iteration.*

Proof. Verification of a polynomial invariant p_t at a program point t is done via fixpoint iteration on sets of generators. Considering a set $G[u]$ of generators representing the ideal $\mathbf{R}^\sharp_{p_t}(u)$ of preconditions at program point u, we know from lemma 5, that an increasing chain of normal-reduced generator systems is bounded by $w \cdot r$. Each time, that the addition of a polynomial p leads to an increase of $G[u]$, the evaluation of $[\![s]\!]^\top(p)$ is triggered for each edge (u, v) labelled with s. Each precondition transformer creates only one precondition polynomial, except for the nondeterministic assignment. Essentially, each $[\![\mathbf{x}_j :=?]\!]^\top$ causes $d+1$ (d the maximal degree of $\mathbf{x}_j$) polynomials to be added to the set of generators at the source u of the corresponding control flow edge. Since the degree d of any variable $\mathbf{x}_j$ in an occurring generator polynomial is bounded by $1.5(w+1)$, we conclude that the total number of increases for the set $G[u]$ of generators for program point u along the control flow edge (u, v) amounts to $\mathcal{O}(w^2 \cdot r)$. As we can estimate the complexity of a complete reduction by $\mathcal{O}(k \cdot r^2)$ with the help of lemma 1, we find that the amount of work induced by a single control flow edge therefore is bounded by $\mathcal{O}(k \cdot w^2 \cdot r^3)$. This provides us with the upper complexity bound stated in this theorem. $\square$

Assume that the maximal degree of a polynomial occurring in an assignment of the input program has degree 2. Then the maximal total degree d of any monomial occurring during fixpoint iteration is bounded by $1.5(w+k) + 3w + 2 = 4.5w + 1.5k + 2$. By prop. 1, the number r of monomials in the complexity estimation of theorem 2 is thus bounded by $(4.5w+k+3)^k$. From that, we deduce that our algorithm runs in polynomial time – at least in case of constantly many variables. Of course, for three variables and $w = 2^5$, the number r of possibly occurring monomials is already beyond $2^{7 \cdot 4} = 2^{28}$ — which is far beyond what one might expect to be practical.

We implemented our approach and evaluated it on selected benchmark programs, similar to the ones from [12]. We considered the series of programs `power-`i which compute sums of $(i-1)$-th powers, i.e., the value $\mathbf{x} = \sum_{\mathbf{y}} \mathbf{y}^{i-1}$. In the case $i = 6$, for example, the invariant $12\mathbf{x} - 2\mathbf{y}^6 - 6\mathbf{y}^5 - 5\mathbf{y}^4 + \mathbf{y}^2$ could be verified for the end point of the program. Additionally, we considered programs `geo-`i for computing variants of the geometrical sum. An overview with verified invariants is shown in table 1. All these invariants could be verified instantly on a contemporary desktop computer with 2.4 GHz and 2GB of main memory.

Table 1. Test programs and verified invariants in $w = 32$

Name	Computation		verified invariant
power-1	$\mathbf{x}_1 = \sum_{k=0}^{K} 1$	$\mathbf{x}_2 = \sum_{k=0}^{K} 1$	$\mathbf{x}_1 = \mathbf{x}_2$
power-2	$\mathbf{x}_1 = \sum_{k=0}^{K} k$	$\mathbf{x}_2 = \sum_{k=0}^{K} 1$	$2\mathbf{x}_1 = \mathbf{x}_2^2 + \mathbf{x}_2$
power-3	$\mathbf{x}_1 = \sum_{k=0}^{K} k^2$	$\mathbf{x}_2 = \sum_{k=0}^{K} 1$	$6\mathbf{x}_1 = 2\mathbf{x}_2^3 + 3\mathbf{x}_2^2 + \mathbf{x}_2$
power-4	$\mathbf{x}_1 = \sum_{k=0}^{K} k^3$	$\mathbf{x}_2 = \sum_{k=0}^{K} 1$	$4\mathbf{x}_1 = \mathbf{x}_2^4 + 2\mathbf{x}_2^3 + \mathbf{x}_2^2$
power-5	$\mathbf{x}_1 = \sum_{k=0}^{K} k^4$	$\mathbf{x}_2 = \sum_{k=0}^{K} 1$	$30\mathbf{x}_1 = 6\mathbf{x}_2^5 - 15\mathbf{x}_2^4 - 10\mathbf{x}_2^3 + \mathbf{x}_2$
power-6	$\mathbf{x}_1 = \sum_{k=0}^{K} k^5$	$\mathbf{x}_2 = \sum_{k=0}^{K} 1$	$12\mathbf{x}_1 = 2\mathbf{x}_2^6 - 6\mathbf{x}_2^5 - 5\mathbf{x}_2^4 + \mathbf{x}_2^2$
geo-1	$\mathbf{x}_1 = (\mathbf{x}_3 - 1)\sum_{k=0}^{K} \mathbf{x}_3^k$	$\mathbf{x}_2 = \mathbf{x}_3^{K-1}$	$\mathbf{x}_1 = \mathbf{x}_2 + 1$
geo-2	$\mathbf{x}_1 = \sum_{k=0}^{K} \mathbf{x}_3^k$	$\mathbf{x}_2 = \mathbf{x}_3^{K-1}$	$\mathbf{x}_1 \cdot (\mathbf{x}_3 - 1) = \mathbf{x}_2\mathbf{x}_3 - 1$
geo-3	$\mathbf{x}_1 = \sum_{k=0}^{K} \mathbf{x}_4 \cdot \mathbf{x}_3^k$	$\mathbf{x}_2 = \mathbf{x}_3^{K-1}$	$\mathbf{x}_1 \cdot (\mathbf{x}_3 - 1) = \mathbf{x}_4\mathbf{x}_3\mathbf{x}_2 - \mathbf{x}_4$

6 Inferring Polynomial Relations over $\mathbb{Z}_{2^w}$

Still, no algorithm is known which, for a given polynomial program, infers all valid polynomial relations over $\mathbb{Q}$. In [9] it is shown, however, that at least all polynomial relations up to a *maximal total degree* can be computed. For the finite ring $\mathbb{Z}_{2^w}$, on the other hand, we know from lemma 2 that every polynomial has an equivalent polynomial of total degree at most $1.5(w + k)$. Therefore over $\mathbb{Z}_{2^w}$, any algorithm which computes all polynomial invariants up to a given total degree is sufficient to compute all valid polynomial invariants.

For a comparison, we remark that, since $\mathbb{Z}_{2^w}$ is finite, the collecting semantics of a polynomial program of length N is finite and computable by ordinary fixpoint iteration in time $N \cdot 2^{\mathcal{O}(wk)}$. Given the set $X \subseteq \mathbb{Z}_{2^w}^k$ of states possibly reaching a program point v, we can determine all polynomials p of total degree at most $1.5(w+k)$ with $p(x) = 0$ for all $x \in X$ by solving an appropriate linear system of $|X| \leq 2^{wk}$ equations for the coefficients of p. For every program point u, this can be done in time $2^{\mathcal{O}(wk)}$. Here, our goal is to improve on this trivial (and intractable) upper bound. Our contribution is to remove the w in the exponent and to provide an algorithm whose runtime, though exponential in k in the worst case, may still be much faster on meaningful examples.

For constructing this algorithm, we are geared to the approach from [9]. This means that we fix a template for the form of polynomials that we want to infer. Such a template is given by a set M of monomials $m = \mathbf{x}_1^{r_1} \ldots \mathbf{x}_k^{r_k}$ with $r_1 + \ldots + r_k \leq 1.5(w + k)$. Note that for small maximal total degree d the cardinality of M is bounded by $(k+1)^d$ while without restriction on d, the cardinality is bounded by an exponential in k (see prop. 1). Given the set M, we introduce a set $\mathbf{A}_M = \{\mathbf{a}_m \mid m \in M\}$ of auxiliary fresh variables $\mathbf{a}_m$ for the coefficients of the monomials m in a possible invariant. The template polynomial p_M for M then is given by $p_M = \sum_{m \in M} \mathbf{a}_m \cdot m$.

Example 5. Consider the program variables $\mathbf{x}_1$ and $\mathbf{x}_2$. Then the template polynomial for the set of all monomials of total degree at most 2 is given by: $\mathbf{a}_1\mathbf{x}_1^2 + \mathbf{a}_2\mathbf{x}_2^2 + \mathbf{a}_3\mathbf{x}_1\mathbf{x}_2 + \mathbf{a}_4\mathbf{x}_1 + \mathbf{a}_5\mathbf{x}_2 + \mathbf{a}_0$. $\square$

With the help of the verification algorithm from section 4, we can compute the weakest precondition for a given template polynomial p_m. Since during fixpoint computation, no substitutions of the generic parameters $\mathbf{a}_m$ are involved, each polynomial p in any occurring set of generators is always of the form $p = \sum_{m \in M} \mathbf{a}_m \cdot q_m$ for polynomials $q_m \in \mathbb{Z}_{2^w}[\mathbf{X}]$. In particular, this holds for the set of generators computed by the fixpoint algorithm for the ideal at the start point st of the program. We have:

Lemma 6. *Assume that G is a set of generators of the ideal $\mathbf{R}^\sharp_{p_t}(\mathsf{st})$ for the template polynomial p_M at program point t. Then for any $a_m \in \mathbb{Z}_{2^w}, m \in M$, the polynomial $\sum_{m \in M} a_m m$ is valid at program point t iff for all $g \in G$, the polynomial $g[a_m/\mathbf{a}_m]_{m \in M}$ is a vanishing polynomial.* □

It remains to determine the values $a_m, m \in M$ for which all polynomials g in a finite set G are vanishing. First assume that the polynomials in G may contain variables from $\mathbf{X}$. Assume w.l.o.g. that it is $\mathbf{x}_k$ which occurs in some polynomial in G where the maximal degree of $\mathbf{x}_k$ in polynomials of G is bounded by d. Then we construct a set G' by:

$$G' = \{g[j/\mathbf{x}_k] \mid g \in G, j = 0, \ldots, d\}$$

The set G' consists of polynomials g' which contain variables from $\mathbf{X} \backslash \{\mathbf{x}_k\}$ only. Moreover by lemma 3, $g[a_m/\mathbf{a}_m]_{m \in M}$ is vanishing for all $g \in G$ iff $g'[a_m/\mathbf{a}_m]_{m \in M}$ is vanishing for all $g' \in G'$. Repeating this procedure, we successively may remove all variables from $\mathbf{X}$ to eventually arrive at a set $\bar{G}$ of polynomials without variables from $\mathbf{X}$. This means each $g \in \bar{G}$ is of the form $g = \sum_{m \in M} \mathbf{a}_m \cdot c_m$ for $c_m \in \mathbb{Z}_{2^w}$. Therefore, we can apply the methods from [11] for *linear* systems of equations over $\mathbb{Z}_{2^w}$ (now with variables from $\mathbf{A}_M$) to determine a set of generators for the $\mathbb{Z}_{2^w}$-module of solutions. Thus, we obtain the following result:

Theorem 3. *Assume p is a polynomial program of length N with k variables over the ring $\mathbb{Z}_{2^w}$. Further assume that M is a subset of monomials of total degree bounded by $1.5(w + k)$. Then all valid polynomial invariants $\sum_{m \in M} c_m m$ with $c_m \in \mathbb{Z}_{2^w}$ can be computed in time $\mathcal{O}(N \cdot k \cdot w^2 (r_0 r)^3)$ where r_0 is the cardinality of M and r is the maximal number of monomials occurring during fixpoint iteration.*

Proof. Generator sets of polynomials over M and X are always composed of polynomials p of the form $p = \sum_{m \in M} \mathbf{a}_m \cdot \mathbf{x}_1^{d_1} \ldots \mathbf{x}_k^{d_k}$. Thus, the number of occurring different monomials is bounded by $r_0 \cdot r$. Therefore, the maximal length of a strictly increasing chain of normal-reduced sets of generators is bounded by $w \cdot r_0 \cdot r$. As the number of monomials in a polynomial is bounded by $r_0 \cdot r$, the costs for updating a normal-reduced set of generators with a single polynomial is now $\mathcal{O}(k \cdot (r_0 \cdot r)^2)$. Again, we have to account $1.5(w+1)$ for the number of polynomials which can be produced by weakest precondition transformers in a single step. Altogether, we therefore have costs $\mathcal{O}(w \cdot w r_0 r \cdot k (r_0 r)^2)$ which are incured at each of the control flow edges of the program to be analysed. □

Finding all valid polynomial invariants means to compute the precondition for a template with all monomials up to degree $d = 1.5(w+k)$. We thus obtain $(1.5(w+k)+1)^k$

as an upper bound for the number r_0 of monomials to be considered in the postcondition. For input programs where the maximal degree of polynomials in assignments or disequalities is bounded by 2, the number r of occurring monomials can be bounded by $(4.5w + k + 3)^k$. Summarising, we find that all polynomial invariants which are valid at a given program point can be inferred by an algorithm whose runtime is only exponential in k. This means that this algorithm is polynomial whenever the number of variables is bounded by a constant.

Example 6. Consider the program `geo-1` next to this paragraph which computes the geometrical sum. At program end, we obtain the invariant $\mathbf{x} - \mathbf{y} + 1 = 0$ as expected. Beyond that, we obtain the additional invariant $2^{31}\mathbf{y} + 2^{31}\mathbf{x} = 0$ which is valid over $\mathbb{Z}_{2^{32}}$ only. This invariant expresses that $\mathbf{x}$ and $\mathbf{y}$ are either both odd or both even at a specific program state. □

```
int count = ?;
int x = 1, y = z;
while (count != 0){
    count = count - 1;
    x = x*z + 1;
    y = z*y;
}
x = x*(z-1);
```

We used a prototypical implementation of the presented approach for conducting a test series whose results on our 2,4 GHz 2 GB machine are shown in table 2. The algorithm quickly terminates when inferring all invariants up to degree i for sums of powers of degree $i - 1$ for $i = 1, 2$ and 3 and also for the two variants of geometrical sums. Interestingly, it failed to terminate within reasonable time bounds for $i = 4$. In those cases when terminating, it inferred the invariants known from the analysis of polynomial relations over $\mathbb{Q}$ — and quite a few extra non-trivial invariants which could not be inferred before. It remains a challenge for future work to improve on our methods so that also more complicated programs such as e.g. `power-4` can be analysed precisely.

Table 2. Test programs and inferred invariants in $w = 32$

Name	**inferred polynomial**	**time**	**space**
power-1	$\mathbf{x}_0 - \mathbf{x}_1$	0.065 sec	51 MB
power-2	$\mathbf{x}_1^2 - 2\mathbf{x}_0 + \mathbf{x}_1$	0.195 sec	63 MB
power-3	$2\mathbf{x}_1^3 - 3\mathbf{x}_1^2 - \mathbf{x}_1 - 6\mathbf{x}_0$, $2^{30}\mathbf{x}_1^2 - 2^{31}\mathbf{x}_0 + 2^{30}\mathbf{x}_1$, $3\cdot 2^{29}\mathbf{x}_0\mathbf{x}_1 + 15\cdot 2^{27}\mathbf{x}_1^2 - 5\cdot 2^{28}\mathbf{x}_0 + 15\cdot 2^{27}\mathbf{x}_1$, $3\cdot 2^{28}\mathbf{x}_0^2 - 25\cdot 1^{26}\mathbf{x}_0\mathbf{x}_1 - 77\cdot 2^{24}\mathbf{x}_1^2 - 25\cdot 2^{25}\mathbf{x}_0 - 77\cdot 2^{24}\mathbf{x}_1$, $21\cdot 2^{24}\mathbf{x}_1^3 + 191\cdot 2^{23}\mathbf{x}_1^2 + 65\cdot 2^{24}\mathbf{x}_0 + 149\cdot 2^{23}\mathbf{x}_1$, $-19\cdot 2^{26}\mathbf{x}_0^3 + 2^{24}\mathbf{x}_0^2\mathbf{x}_1 - 235\cdot 2^{22}\mathbf{x}_0\mathbf{x}_1^2 - 191\cdot 2^{23}\mathbf{x}_0\mathbf{x}_1 + 57\cdot 2^{25}\mathbf{x}_1^2 - 27\cdot 2^{26}\mathbf{x}_0 + 37\cdot 2^{25}\mathbf{x}_1$	1.115 sec	89 MB
power-4	n.a.	>24 h	> 1 GB
geo-1	$\mathbf{x}_0 - \mathbf{x}_1 - 1$, $2^{31}\mathbf{x}_1 + 2^{31}\mathbf{x}_2$	0.064 sec	48 MB
geo-2	$2^{31}\mathbf{x}_1\mathbf{x}_2 + 2^{31}\mathbf{x}_2$, $2^{31}\mathbf{x}_1 + 2^{31}\mathbf{x}_2$, $2^{28}\mathbf{x}_1^2 + 230\mathbf{x}_1\mathbf{x}_2 - 7\cdot 2^{28}\mathbf{x}_2^2 - 3\cdot 2^{29}\mathbf{x}_2 + 2^{31}$, $\mathbf{x}_0\mathbf{x}_2 - \mathbf{x}_1\mathbf{x}_2 - \mathbf{x}_0 + 1$	0.636 sec	65 MB
geo-3	*23 polynomials …*	2.064 sec	96 MB

7 Conclusion

We have shown that verifying polynomial program invariants over $\mathbb{Z}_{2^w}$ is *PSPACE*-complete. By that, we have provided a clarification of the complexity of another analysis problem in the taxonomy of [6]. Beyond the theoretical algorithm for the upper bound, we have provided a realistic method by means of normal-reduced generator sets. In case of constantly many variables, this algorithm runs in polynomial time. Indeed, our prototypical implementation was amazingly fast on all tested examples.

We extended the method for checking invariants to a method for inferring polynomial invariants of bounded degree — which in case of the ring $\mathbb{Z}_{2^w}$ also allows to infer all polynomial invariants. Beyond the vanishing polynomials, the algorithm finds further invariants over $\mathbb{Z}_{2^w}$, which would not be valid over the field $\mathbb{Q}$ and thus cannot be detected by any analyser over $\mathbb{Q}$. While still being polynomial for constantly many variables, our method turned out to be decently efficient only for small numbers of variables and low degree invariants. It remains for future work to improve on the method for inferring invariants in order to deal with larger numbers of variables and moderate degrees at least for certain meaningful examples.

References

1. Becker, T., Weispfenning, V.: Gröbner Bases – a computational approach to commutative algebra. Springer, New York (1993)
2. Colon, M.: Polynomial approximations of the relational semantics of imperative programs. Science of Computer Programming 64, 76–96 (2007)
3. Hungerbühler, N., Specker, E.: A generalization of the smarandache function to several variables. Integers: Electronic Journal Combinatorial Number Theory 6 (2006)
4. Karr, M.: Affine Relationships Among Variables of a Program. Acta Informatica 6, 133–151 (1976)
5. Matsumura, H.: Commutative Ring Theory. Cambridge University Press, Cambridge (1989)
6. Müller-Olm, M., Rüthing, O.: On the complexity of constant propagation. In: Sands, D. (ed.) ESOP 2001 and ETAPS 2001. LNCS, vol. 2028, Springer, Heidelberg (2001)
7. Müller-Olm, M., Seidl, H.: Polynomial Constants are Decidable. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 4–19. Springer, Heidelberg (2002)
8. Müller-Olm, M., Petter, M., Seidl, H.: Interprocedurally analyzing polynomial identities. In: Durand, B., Thomas, W. (eds.) STACS 2006. LNCS, vol. 3884, pp. 50–67. Springer, Heidelberg (2006)
9. Müller-Olm, M., Seidl, H.: Computing Polynomial Program Invariants. Information Processing Letters (IPL) 91(5), 233–244 (2004)
10. Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444. Springer, Heidelberg (2005)
11. Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. ACM Transactions on Programming Languages and Systems (TOPLAS) 29(5) (2007)
12. Petter, M.: Berechnung von polynomiellen Invarianten. Master's thesis, Technische Universität München, München (2004)
13. Rodríguez-Carbonell, E., Kapur, D.: An abstract interpretation approach for automatic generation of polynomial invariants. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, Springer, Heidelberg (2004)

14. Rodríguez-Carbonell, E., Kapur, D.: Automatic generation of polynomial invariants of bounded degree using abstract interpretation. Science of Computer Programming 64, 54–75 (2007)
15. Sankaranarayanan, S., Henry, Z.M., Sipma, B.: Non-linear loop invariant generation using gröbner bases. In: ACM SIGPLAN Principles of Programming Languages (POPL) (2004)
16. Shekhar, N., Kalla, P., Enescu, F., Gopalakrishnan, S.: Equivalence verification of polynomial datapaths with fixed-size bit-vectors using finite ring algebra. In: International Conference on Computer-Aided Design (ICCAD), pp. 291–296 (2005)
17. Singmaster, D.: On polynomial functions (mod m). Journal of Number Theory 6, 345–352 (1974)
18. Wienand, O.: The Groebner basis of the ideal of vanishing polynomials. Journal of Symbolic Computation (JSC) (to appear, 2007); Preprint in arxiv math.AC/0801.1177

Splitting the Control Flow with Boolean Flags

Axel Simon

DI, École Normale Supérieure, rue d'Ulm, 75230 Paris cedex 05, France[⋆]
Axel.Simon@ens.fr

Abstract. Tools for proving the absence of run-time errors often deploy a numeric domain that approximates the possible values of a variable using linear inequalities. These abstractions are adequate since the correct program state is often convex. For instance, if the upper and lower bound of an index lie within the bounds of an array, then so do all the indices inbetween. In certain cases, for example when analysing a division operation, the correct program state is not convex. In this case correctness can be shown by splitting the control flow path, that is, by partitioning the set of execution traces which is normally implemented by analysing a path several times. We show that adding a Boolean flag to the numeric domain has the same effect. The paper discusses prerequisites, limitations and presents an improved points-to analysis using Boolean flags.

1 Introduction

With better expressiveness and scalability, static analysis is increasingly used to prove the absence of run-time errors of certain types of C software [3,7]. A classic approach of approximating the state at a particular point in the program are convex spaces described by a finite conjunction of linear inequalities (a polyhedron [8]). For example, an array access `a[i]` in C is within bounds if the inferred set of inequalities at that program point imply a space where $i \geq 0$ and $i \leq s-1$ where s is the size of the array. In certain cases, a single convex approximation to the possible state space is not precise enough. A notorious example is a guard against a division by zero, as implemented by the following C fragment:

```
int r=MAX_INT;
if (d!=0) r=v/d;
```

The test $d \neq 0$ must be implemented by intersecting the current state, say P, with $d > 0$ and $d < 0$, resulting in $P^+ = P \cap [\![d > 0]\!]$ and $P^- = P \cap [\![d < 0]\!]$ where $[\![c]\!]$ denotes the sub-space of the Euclidean space that satisfies the constraint c. The state with which the division operation is analysed is approximated by $P' = P^+ \sqcup P^-$, where $\sqcup$ is the join of the lattice, implemented by calculating the closure of the convex hull of the two polyhedra. However, the convex hull

[⋆] This work was supported by the INRIA project "Abstraction" of CNRS and ENS.

© Springer-Verlag Berlin Heidelberg 2008

re-introduces the state $P \sqcap \{d = 0\}$ for which the division `r=v/d` is erroneous. In fact, it turns out that $P' = P$ whenever P^+ and P^- are non-empty.

The Astrée analyser verifies [7] the above example by partitioning the set of execution traces, that is, by executing the division instructions twice, first with P^-, then with P^+. The question when to partition the traces into two sets and when to rejoin the separate trace sets is guided by heuristics and by manual annotations [14]. Thus, the Astrée analyser is designed with the ambition to choose the correct split and joint points in order to prove the program correct.[1] The Static Driver Verifier (SDV) [3] verifies the API usage of device drivers by converting their C source into a program with only Boolean variables [5] whose correctness is then checked against a pre-defined set of rules using a model checker [4]. In case a rule cannot be proved or disproved, a new predicate is synthesised, thereby splitting the set of traces in the concrete program with the aim to improve the precision of the abstraction with respect to the property of interest. Thus, SDV iteratively splits the control flow path and re-analyses the source until the rules can be proved. Both approaches explicitly partition the set of traces or, semantically equivalent, perform a program transformation that splits the control flow path into two. The cost of the analysis thus depends on the choice of the split and join points. In this work we propose to use polyhedral analysis but implement the split of a control flow path by adding a Boolean flag to the domain. Although the cost of adding such a flag might be higher than analysing the code twice, an additional flag is very cheap if the two states are identical. This allows our method to be used opportunistically, that is, Boolean flags may be introduced by default when it is not know if two states need to be separated later. Furthermore, our approach is appealing due to its simplicity.

The motivation of our work stems from the need to improve points-to information during a polyhedral value-range analysis. By using Boolean flags to indicate if an l-value is part of a points-to set, it is possible to infer that e.g. the error flag returned from a function is set whenever a pointer is `NULL`. Points-to information presents a good example where the re-analysis of code with every possible set of points-to information can be more expensive than adding flags.

Adding a Boolean flag may result in a loss of precision, especially when used with weakly relational domain such as Octagons [15] or TVPI polyhedra [22]. To summarise, this paper makes the following contributions:

– We show that, under certain prerequisites, adding a Boolean flag is equivalent to analyzing a control flow path twice.
– We discuss the way Boolean flags can be used in the context of general polyhedra and weakly relational domains such as the TVPI domain.
– We show how points-to analysis and numeric analysis can be combined using Boolean flags, thereby improving precision without explicit propagation between the two domains.

[1] Astrée can also partition the set of traces by the value of variables. In this case, joining the partitions is automatic, e.g. when the variable goes out of scope. Our proposal can be used for both applications but is more amenable to the latter task.

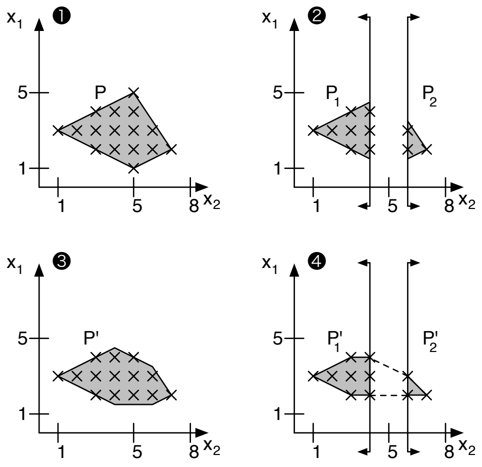


Fig. 1. Example to show that $\mathbb{Z}$-polyhedra are not closed under intersection. Integral points are shown as crosses and the half-spaces $[\![x \leq 4]\!]$ and $[\![x \geq 6]\!]$ are indicated by vertical lines with arrows pointing towards the feasible space.

The paper is organised as follows. The next section gives a definition of the polyhedral domain before Sec. 3 discusses the separation of states using Boolean flags. Section 4 demonstrates practical applications and Section 5 concludes.

2 An Introduction to Convex Polyhedra

This section introduces some basic notation and properties of polyhedra, putting particular emphasis on the set of integral points contained in a polyhedron.

A polyhedral analysis expresses numeric constraints over the set of abstract variables $\mathcal{X}$. For the sake of this section, let $\boldsymbol{x}$ denote the vector of all variables in $\mathcal{X}$, thereby imposing an order on $\mathcal{X}$. Let *Lin* denote the set of linear expressions of the form $\boldsymbol{a} \cdot \boldsymbol{x}$ where $\boldsymbol{a} \in \mathbb{Z}^{|\mathcal{X}|}$ and let *Ineq* denote the set of linear inequalities $\boldsymbol{a} \cdot \boldsymbol{x} \leq c$ where $c \in \mathbb{Z}$. Let e.g. $6x_3 \leq x_1 + 5$ abbreviate $\langle -1, 0, 6, 0, \ldots 0\rangle \cdot \boldsymbol{x} \leq 5$ and let e.g. $x_2 = 7$ abbreviate the two opposing inequalities $x_2 \leq 7$ and $x_2 \geq 7$, the latter being an abbreviation of $-x_2 \leq -7$. For simplicity, we assume that the analysis only infers integral properties and we use the notation $e_1 < e_2$ to abbreviate $e_1 \leq e_2 - 1$. Each inequality $\boldsymbol{a} \cdot \boldsymbol{x} \leq c \in \mathit{Ineq}$ induces a half-space $[\![\boldsymbol{a} \cdot \boldsymbol{x} \leq c]\!] = \{\boldsymbol{x} \in \mathbb{Q}^{|\mathcal{X}|} \mid \boldsymbol{a} \cdot \boldsymbol{x} \leq c\}$. A set of inequalities $I \subseteq \mathit{Ineq}$ induces a closed, convex space $[\![I]\!] = \bigcap_{\iota \in I} [\![\iota]\!]$. Define $\mathit{Poly} = \{[\![I]\!] \mid I \in \mathit{Ineq} \wedge |I| \in \mathbb{N}\}$ to be the set of all closed, convex polyhedra. Let $P_1, P_2 \in \mathit{Poly}$, then $P_1 \sqsubseteq P_2$ iff $P_1 \subseteq P_2$ and define $P_1 \sqcap P_2$ as $P_1 \cap P_2$ which can be implemented by joining the two sets of inequalities that describe P_1 and P_2. Furthermore define $P_1 \sqcup P_2$ to be the smallest polyhedron P such that $P_1 \sqsubseteq P$ and $P_2 \sqsubseteq P$, that is $P_1 \sqcup P_2 = \bigcap\{P \in \mathit{Poly} \mid P_1 \sqsubseteq P \wedge P_2 \sqsubseteq P\}$. This operation corresponds to the topological closure of the convex hull of P_1 and P_2. Since the number of inequalities $|I|$ defining a

$P \in \mathit{Poly}$ is finite by definition, the lattice of convex polyhedra $\langle \mathit{Poly}, \sqsubseteq, \sqcup, \sqcap \rangle$ is incomplete as neither the join nor meet of an arbitrary number of polyhedra is necessarily a polyhedron. Thus, in order to ensure that a fixpoint calculation always terminates on elements of *Poly* a widening operator is required [8].

In the context of storing Boolean flags in a polyhedral domain, it is interesting to equate polyhedra that contain the same set of integral points. These equivalence classes define the lattice of $\mathbb{Z}$-polyhedra $\langle \mathit{Poly}_{\equiv\mathbb{Z}}, \sqsubseteq^{\mathbb{Z}}, \sqcup^{\mathbb{Z}}, \sqcap^{\mathbb{Z}} \rangle$. Each equivalence class can be represented by its smallest polyhedron, namely a $\mathbb{Z}$-polyhedron which is characterised by the fact that all its vertices, that is, all points that are not a convex combination of other points, have integral coordinates. In this case it is possible to set $\sqsubseteq^{\mathbb{Z}} = \sqsubseteq$ and $\sqcup^{\mathbb{Z}} = \sqcup$. However, the meet operation $\sqcap$ is not closed for $\mathbb{Z}$-polyhedra. In order to illustrate this, consider Fig. 1. The state space $P \in \mathit{Poly}_{\equiv\mathbb{Z}}$ over x_1, x_2 in the first graph is transformed by evaluating the conditional $x_2 \neq 5$ which is implemented by calculating $P' = (P \sqcap [\![x \leq 4]\!]) \sqcup (P \sqcap [\![x \geq 6]\!])$. Observe that the input P as well as the two half-spaces $[\![x \leq 4]\!]$ and $[\![x \geq 6]\!]$ are $\mathbb{Z}$-polyhedra. The second graph shows the two intermediate results $P_1 = P \sqcap [\![x \leq 4]\!]$ and $P_2 = P \sqcap [\![x \geq 6]\!]$ which both have two non-integral vertices. As a consequence, the join of P_1 and P_2, shown as third graph, has non-integral vertices as well and is therefore not a $\mathbb{Z}$-polyhedron. However, if the intermediate results were shrunk around the contained integral point sets, as done in the fourth graph, all vertices of the intermediate results would be integral and the join would be a $\mathbb{Z}$-polyhedron, too. However, for general, n-dimensional polyhedra, the number of inequalities necessary to represent a $\mathbb{Z}$-polyhedron can grow exponentially with respect to a polyhedron over $\mathbb{Q}$ that contains the same integral points [17, Chap. 23]. Thus, no efficient algorithm exists to implement the $\sqcap$-operation on $\mathbb{Z}$-polyhedra. For the weakly relational domain of TVPI polyhedra where each inequality takes on the form $ax + by \leq c$ where $a, b, c \in \mathbb{Z}$, reducing the polyhedron to a $\mathbb{Z}$-polyhedron is still NP-complete [13], however, a polynominal algorithm exists for calculating the $\mathbb{Z}$-polyhedron for any given planar polyhedron [11] on which an efficient approximation of $\mathbb{Z}$-TVPI polyhedra can be built. Interestingly, for the Octagon domain [15], an efficient algorithm exists [2].

The next section re-examines the introductory example and illustrates its analysis in the context of using a single convex polyhedron.

3 Principles of Boolean Flags in Polyhedra

Reconsider the evaluation of the following code fragment, given the state P:

```
int r=MAX_INT;
if (d!=0) r=v/d;
```

Suppose this block of code is executed with a value of $[-9, 9]$ for `d`. Rather than analysing the division twice, once with positive values of d, once with negative values of d, Fig. 2 shows how the states P^+ and P^- can be stored in a single

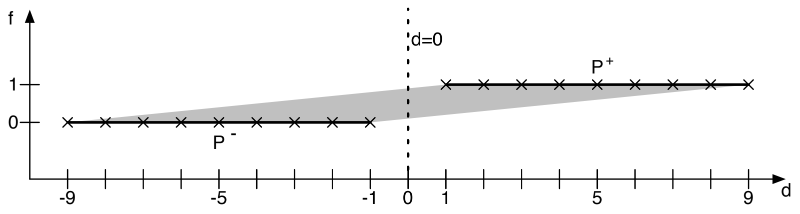


Fig. 2. Using a Boolean flag to perform control flow splitting. Feasible integral points are indicated by crosses, the dashed line indicating the polyhedron $[\![d = 0]\!]$.

state without introducing an integral point where $d = 0$. Specifically, the figure shows the state $(P^- \sqcap [\![f = 0]\!]) \sqcup (P^+ \sqcap [\![f = 1]\!])$ which collapses to the empty $\mathbb{Z}$-polyhedron when intersected with $d = 0$, where $[\![d = 0]\!]$ is the state for which the division is erroneous. The prerequisites for merging two states without loss is that both states are represented by polytopes (polyhedra in which all variables are bounded) and that techniques for integral tightening are present. This is formalised in the following proposition:

Proposition 1. *Let* $P_0, P_1 \in \mathit{Poly}$*, let* $P = (P_0 \sqcap [\![f = 0]\!]) \sqcup (P_1 \sqcap [\![f = 1]\!])$ *and* $P'_i = P \sqcap [\![f = i]\!]$ *for* $i = 0, 1$*. Then* $P_i \sqcap [\![f = i]\!] \cap \mathbb{Z}^n = P'_i \cap \mathbb{Z}^n$ *for all* $i = 0, 1$*.*

Proof. Without loss of generality, assume $i = 0$. Suppose that $\mathcal{X}$ is arranged as $\langle x_1, \ldots x_{n-1}, f\rangle = \boldsymbol{x}$ which we abbreviate as $\langle \bar{\boldsymbol{x}}|f\rangle = \boldsymbol{x}$. We consider two cases:

"soundness": Let $\langle \bar{\boldsymbol{a}}|f\rangle \in P_0 \sqcap [\![f = 0]\!] \cap \mathbb{Z}^n$. Then $\langle \bar{\boldsymbol{a}}|f\rangle \in P$ by the definition of $\sqcup$. Since $f = 0$ in $\langle \bar{\boldsymbol{a}}|f\rangle$ it follows that $\langle \bar{\boldsymbol{a}}|f\rangle \in P \sqcap [\![f = 0]\!]$ and thus $\langle \bar{\boldsymbol{a}}|f\rangle \in P'_0$. We chose the vector such that $\langle \bar{\boldsymbol{a}}|f\rangle \in \mathbb{Z}^n$ and hence $\langle \bar{\boldsymbol{a}}|f\rangle \in P'_0 \cap \mathbb{Z}^n$.

"completeness": Let $\langle \bar{\boldsymbol{a}}|f\rangle \in P'_0 \cap \mathbb{Z}^n$. Then $\langle \bar{\boldsymbol{a}}|f\rangle \in P \sqcap [\![f = 0]\!]$ and hence $f = 0$. For the sake of a contradiction, suppose that $\langle \bar{\boldsymbol{a}}|0\rangle \notin P_0$. Since $P \in \mathit{Poly}$ is convex, $\langle \bar{\boldsymbol{a}}|0\rangle = \lambda\langle \bar{\boldsymbol{a}}_{\boldsymbol{1}}|f_1\rangle + (1 - \lambda)\langle \bar{\boldsymbol{a}}_{\boldsymbol{2}}|f_2\rangle$ for some $0 \le \lambda \le 1$. Observe that the join $P = (P_0 \sqcap [\![f = 0]\!]) \sqcup (P_1 \sqcap [\![f = 1]\!])$ is defined as an intersection of all states $\hat{P}$ such that $(P_i \sqcap [\![f = i]\!]) \sqsubseteq \hat{P}$ for $i = 0, 1$; this holds in particular for $\hat{P} = [\![f \ge 0]\!]$ and for $\hat{P} = [\![f \le 1]\!]$. Thus, $0 \le f \le 1$ for all $\langle \bar{\boldsymbol{a}}|f\rangle \in P$. Hence, $0 \le f_1 \le 1$ and $0 \le f_2 \le 1$ must hold. Given the constraints $0 = \lambda f_1 + (1 - \lambda) f_2$ and $\bar{\boldsymbol{a}} = \lambda\bar{\boldsymbol{a}}_{\boldsymbol{1}} + (1 - \lambda)\bar{\boldsymbol{a}}_{\boldsymbol{2}}$, either $f_1 = 0$ or $f_2 = 0$ so that one vector, say $\langle \bar{\boldsymbol{a}}_{\boldsymbol{1}}|f_1\rangle$, must lie in P_0, which implies $\lambda = 1$. In particular with $\bar{\boldsymbol{a}} = \lambda\bar{\boldsymbol{a}}_{\boldsymbol{1}} + (1 - \lambda)\bar{\boldsymbol{a}}_{\boldsymbol{2}}$ and $\bar{\boldsymbol{a}}_{\boldsymbol{2}} \in \mathbb{Z}^{n-1}$ having only finite coefficients, $\bar{\boldsymbol{a}} = \bar{\boldsymbol{a}}_{\boldsymbol{1}}$ follows, which contradicts our assumption of $\langle \bar{\boldsymbol{a}}|0\rangle \notin P_0$.

Equivalently, *Poly* over $\mathcal{X} \cup \{f\}$ is isomorphic to $\mathit{Poly}^{\{0,1\}}$ over $\mathcal{X}$, a reduced cardinal power domain [6, Sect. 10.2] and thus equivalent to a partitioning. We briefly comment on the requirements that P_0, P_1 must be polytopes and integral.

3.1 Boolean Flags and Unbounded Polyhedra

With respect to the first requirement, namely that the state space described by the polyhedron must be bounded, Fig. 3 shows that a precision loss occurs for

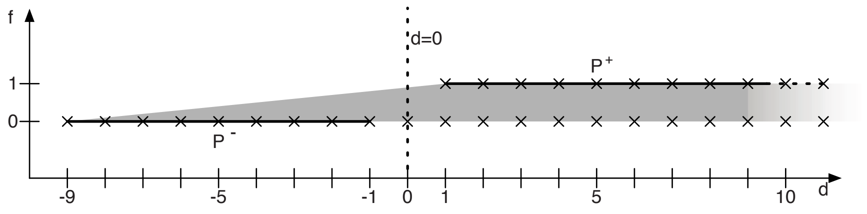


Fig. 3. Adding a Boolean flag to an unbounded polyhedra results in a loss of precision

unbounded polyhedra. Specifically, taking convex combinations of points in the state $P^+ = [\![\{d \geq 1, f = 1\}]\!]$ and those in $P^- = [\![\{-9 \leq d \leq -1, f = 0\}]\!]$ leads to the grey state. Even though the line $[\![\{f = 0, d > -1\}]\!]$ is not part of the grey state, the polyhedral join $P^+ \sqcup P^-$ approximates this state with sets of non-strict inequalities, thereby including the line. Hence, the definition of $\sqcup$ automatically closes the resulting space and thereby introduces points $\langle \bar{\boldsymbol{a}}, f\rangle \in P^+ \sqcup P^-$ where $f = 0$, even though $\langle \bar{\boldsymbol{a}}, 0\rangle \notin P^+$. As a result the intersection with $[\![d = 0]\!]$ contains $\langle 0, 0\rangle \in \mathbb{Z}^2$ and the verification of the division operation fails. Note that allowing strict inequalities $\boldsymbol{a} \cdot \boldsymbol{x} < c$ to describe open facets [1] is not sufficient to define the convex space $\bigcap\{S \mid P^+ \subseteq S \wedge P^- \subseteq S\}$ as the lower bound on f is closed for $-9 \leq d \leq -1$ and open for $d > -1$.

The imprecise handling of unbounded polyhedra is generally not a problem in verifiers that perform a forward reachability analysis as program variables are usually finite and wrap when they exceed their limit. Flagging wrapping as erroneous [7] or making wrapping explicit [21] effectively restricts the range of a variable. Thus, states in a forward analysis are usually bounded. Polyhedra have also been used to infer an input-output relationship of a function, thereby achieving a context-sensitive analysis by instantiating this input-output behaviour at various call sites [10]. In this application the inputs are generally unbounded and a Boolean flag does not distinguish any differences between states. However, even in this application it might be possible to restrict the range of input variables to the maximum range that the concrete program variable may take on, thereby ensuring that input-output relationships are inferred using polytopes.

3.2 Integrality of the Solution Space

A second prerequisite for distinguishing two states within a single polyhedron is that the polyhedron is reduced to the contained $\mathbb{Z}$-polyhedron upon each intersection. Tightening a polyhedron to a $\mathbb{Z}$-polyhedron is an exponential operation which can be observed by translating a Boolean function f over n variables to a $\mathbb{Z}$-polyhedron over $\mathbb{Z}^n$ by calculating the convex hull of all Boolean vectors (using 0,1 for false and true) for which f is true. For example, Fig. 4 shows how common Boolean functions over two variables are represented as planar polyhedra. An argument similar to Prop. 1 is possible to show that joining all n-ary vectors for which f is true leads to a polyhedron that expresses f exactly. The integral meet operation $\sqcap^{\mathbb{Z}}$ therefore becomes a decision procedure for satisfiability of n-ary

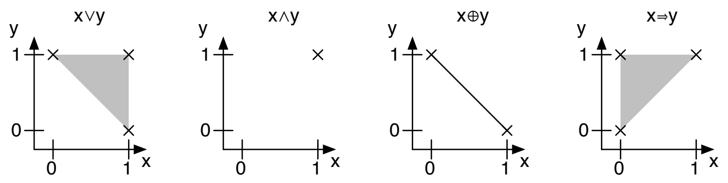


Fig. 4. Boolean functions can be expressed exactly in the polyhedral domain when satisfiable assignments of variables are modelled as vertices in the polyhedron

Boolean formulae. Hence, Octagons [15] together with the complete algorithm for $\sqcap^{\mathbb{Z}}$ presented in [2] provides an efficient decision procedure for 2-SAT.

While calculating a full $\mathbb{Z}$-polyhedron from a given polyhedron with rational intersection points is expensive, a cheap approximation often suffices in practice. For instance, the abstract transfer function of the division operation will add the constraint $d = 0$ to the state space shown in Fig. 2 with the result that possible values of f lie in $[0.1, 0.9]$. Rounding the bounds to the nearest feasible integral value yields the empty interval $[1, 0]$ which indicates an unreachable state, thereby proving that a division by zero cannot happen.

3.3 Using Boolean Flags in Common Polyhedral Domains

In this section we briefly discuss the applicability of our approach to three polyhedral domains: general convex polyhedra, TVPI polyhedra and Octagons.

General convex polyhedra [8] over $\mathbb{Q}^n$ can be defined by sets of inequalities of the form $a_1x_1 + \ldots a_nx_n \leq c$ where $a_1, \ldots a_n, c \in \mathbb{Z}$. The major hindrance for a widespread use is the complexity of the join operation $P_1 \sqcup P_2$ which is commonly implemented by converting the set of inequalities describing P_1 and P_2 into the vertex, ray and line representation which is usually exponential in the number of inequalities, even for simple polyhedra. This intermediate representation can be avoided by using projection to calculate the convex hull [20]. However, we are not aware of any efficient methods to approximate the calculation of a $\mathbb{Z}$-polyhedron from a rational polyhedron. One simple step is to calculate the common divisor of the coefficients $d = \gcd(a_1, \ldots a_n)$ and to tighten each inequality to $\frac{a_1}{d}x_1 + \ldots \frac{a_n}{d}x_n \leq \lfloor\frac{c}{d}\rfloor$. Rounding the constant down looses no integral solutions and thus constitutes a simple integral tightening [16].

The TVPI domain [22] is a weakly relational domain in that it only tracks relationships of the form $ax_i + bx_j \leq c$ where $a, b, c \in \mathbb{Z}$ and $x_i, x_j \in \mathcal{X}$. The idea is to repeatedly calculate resultants, that is, to combine any two inequalities $ax + by \leq c_{xy}$ and $dy + ez \leq c_{yz}$ where $sgn(b) = -sgn(d)$ to a new inequality $fx + gz \leq c_{xz}$ and to add this inequality to the set unless it is redundant. Once no more non-redundant inequalities can be generated, the system is called *closed*. On a closed system, the join $\sqcup$ and inclusion test $\sqsubseteq$ on the n-dimensional TVPI polyhedron can be implemented by performing a much simpler operation on $O(n^2)$ planar polyhedra. Specifically, the planar algorithms are run on

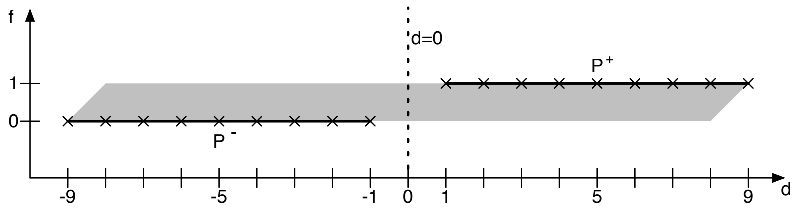


Fig. 5. The shown state is the octahedral approximation of the state in Fig. 2

the inequalities that only contain x_i, x_j, for each pair of variables $x_i, x_j \in \mathcal{X}$. Each planar operation runs in at most $O(m \log m)$ where m is the number of inequalities, thus providing an efficient domain. Furthermore, integral tightening is possible by calculating the $\mathbb{Z}$-polyhedron for each projection in $O(m \log |A|)$ where A is the largest coefficients in any of the inequalities [11]. However, after applying this algorithm to the projection x_i, x_j, it is possible to calculate new resultants by combining the new inequalities with other inequalities containing x_i or x_j which may create rational intersection points in other, already tightened, projections. In fact, calculating a $\mathbb{Z}$-TVPI polyhedron cannot be done efficiently, as the problem is NP-complete [13]. However, performing integral tightening once after each closure is an efficient approximation to a $\mathbb{Z}$-TVPI polyhedron.

Since the TVPI domain has to approximate every inequality with more than two variables, the ability to separate states within the same TVPI polyhedron using a Boolean flag is somewhat limited. In fact, a Boolean flag f in the TVPI domain can only state that the range of a program variable is different. For instance, the TVPI domain is able to express the polyhedron in Fig. 2 and thereby prove the introductory example correct. It is beyond the TVPI domain to state how the linear relationship between two variables change between two states. This is discussed in Sect. 5 where we also suggest a partial solution.

The third domain we consider is the Octagon domain. This domain pre-dates the TVPI domain and is based on similar ideas [15]. The Octagon domain can express inequalities of the form $\pm x_i \pm x_j \leq c$ where $c \in \mathbb{Q}$ or $c \in \mathbb{Z}$. A variant of the Floyd-Warshall shortest paths algorithm is used to calculate a closed system. As above, calculating the join and the inclusion check is implemented for each x_i, x_j variable pair and is as simple as calculating the maximum and performing a comparison on the constants, respectively. Recently, it was shown how the closure algorithm can be modified to obtain a $\mathbb{Z}$-Octagon [2]. However, due to the restriction on the coefficients of the Octagon domain to $+1$ and -1, adding a Boolean flag cannot generally separate two ranges within a single Octagon. Specifically, a Boolean flag can only express the change of another variable by one or minus one, any other change is approximated. Figure 5 shows the state space $P^+ \sqcup P^-$ of the introductory example when approximated with the Octagon domain. Given that the state space can only be delimited by inequalities with unit coefficients, the approximation includes the points $\langle d, f \rangle = \langle 0, 0 \rangle$ and $\langle 0, 1 \rangle$ and hence cannot show that $d \neq 0$. Thus, in the context of the Octagon domain, splitting the set of traces along a control flow path can only be attained by re-evaluating the path several times [14], an approach that is abundant in the Astrée analyser [7] whose principal relational domain is the Octagon domain.

4 Applications of Control Flow Path Splitting

In the remainder of the paper we present various applications where Boolean flags are a valuable alternative to a repeated analysis. We commence with an example on pointer analysis before discussing Boolean flags in string buffer analysis [19].

4.1 Refining Points-To Analysis

The verification of C code often hinges on the ability to analyse pointer operations precisely. A cheap approach to dealing with pointers is to run a flow-insensitive points-to analysis [12] up front and assume that in each pointer access all memory regions in the points-to set of that pointer are accessed. A more accurate approach is to perform a flow-sensitive points-to analysis alongside the fixpoint computation on the numeric domain. To this end, define the points-to domain $Pts = \mathcal{X} \to \mathcal{P}(\mathcal{A})$ which maps each abstract variable $x \in \mathcal{X}$ to a set of abstract addresses $\{a_1, \ldots a_k\} \in \mathcal{P}(\mathcal{A})$ where each abstract address a_i represents the address of a global, automatic or one or more heap-allocated memory regions. The analysis now calculates a fixpoint over the product domain $\langle P, A\rangle \in Poly \times Pts$. This flow-sensitive analysis allows points-to sets to be refined through conditionals and, in particular, it can track if a pointer is `NULL`.

In order to define the semantics of points-to sets, we define $\rho : \mathcal{A} \to \mathcal{P}(\mathbb{N})$ to map an abstract address to concrete addresses. Using this map makes it possible to model variables in functions that are currently not executed as well as summarised heap regions that are represented with a single abstract variable. The possible values of a concrete variable `p` when modelled by x_p are defined as

$$\gamma_{\mathtt{p}}(\langle P, A\rangle) := \{v + p \mid v \in P(x_p) \wedge p \in \rho(a) \wedge a \in A(x_p)\},$$

where $P(x_p)$ denotes the set of integral values that the variable x_p can take on in $P \in Poly$. In order to state that a pointer may be `NULL` or is simply a pure value, we introduce the special abstract address $\textsc{null}$ with $\rho(\textsc{null}) = \{0\}$. As an example, consider the variable `p` that is represented by $x_p \in \mathcal{X}$ with $P(x_p) = [0, 1]$ and $A(x_p) = \{\textsc{null}, a_1, a_2\}$. Assuming that the abstract addresses map to the addresses $\rho(a_1) = \{0x4000\}$ and $\rho(a_2) = \{0x4004\}$, the program variable `p` can take on the values $0, 1, 0x4000, 0x4001, 0x4004$ and $0x4005$.

Operations on the product domain $\langle P, A\rangle \in Poly \times Pts$ have to evaluate the information in both domains and update them accordingly. For instance, the expression `p+q-r`, where `p` is represented by $x_p \in \mathcal{X}$ and so on for `q` and `r`, has different values, depending on the points-to sets $A(x_p)$, $A(x_q)$ and $A(x_r)$. For instance, if $\{a\} = A(x_q)$ and $\{a\} = A(x_r)$ then `q-r` represents a pointer difference, its value is $x_p + x_q - x_r$ and its points-to set is $A(x_p)$. However, if $|A(x_r)| \geq 2$ a severe loss of precision occurs: Suppose $A(x_r) = A(x_q) = \{a_1, a_2\}$ then it is not clear that the pointers contain the same abstract address and it has to be assumed that $a_1 \in A(x_r)$, $a_2 \in A(x_q)$ is possible. A sound approximation is to return the points-to set $\{\text{NULL}\}$ and $[0, 2^{32}-1]$ for the value of the expression.

Applying the Abstraction. In order to illustrate the weakness of the above abstraction, consider the following call to the function `f` that may assign to `p`:

```
char *p;
int r;
r = f(&p);
/* other statments here */
if (r) printf("value:␣%s", p);
```

The return value of `f` indicates whether setting `p` was successful; if it was, the value is printed. The function `f` itself is implemented as follows:

```
int f(char** pp) {
  if (rand()) return 0; /* error */
  *pp = "Success."; return 1; /* success */
}
```

Since the value of `p` is not initialised, the corresponding variable x_p has the points-to set $A(x_p) = \{\text{NULL}\}$ and takes on the range $[0, 2^{32}-1]$, assuming `p` is a 32-bit variable. In case the random number generator returns zero, the pointer is set to the address of the string buffer `"Success."` with an offset of zero. Hence, at the end of `f`, the value of x_p is zero and its points-to set is $A(x_p) = \{a_s\}$ where a_s denotes the address of the string. By returning a different value for each case, the range of the variable x_p is $[0, 2^{32} - 1]$ whenever $r = 0$ and $[0, 0]$ whenever $r = 1$. Thus, testing the return flag in the caller will restrict the offset x_p of the pointer to zero before `printf` is called. However, the points-to set remains unchanged and thus, the analysis will warn that `p` might be `NULL`.

This example can readily be verified by partitioning the set of traces at the `if` statement in the function `f` such that $P(x_p) = [0, 2^{32} - 1]$ and $A(x_p) = \{\text{NULL}\}$ in one set of traces and $P(x_p) = [0, 0]$ and $A(x_p) = \{a_s\}$ in the other. However, it is also possible to use a Boolean flag f_p^s to indicate if x_p contains the address a_s. In particular, the analysis will infer that f_p^s is equal to `r`. Tracking a single equality constraint in addition to the polyhedron of the above analysis is likely to be cheaper than analysing a potentially long sequence of statements twice.

A Revised Abstraction. In order to track the contents of points-to sets in the numeric part of the domain, we use a revised points-to domain $A : \mathcal{X} \to \mathcal{P}(\mathcal{A} \times \mathcal{X})$ that is fixed such that $A(x_p) = \{\langle a_1, f_1^p\rangle, \ldots \langle a_k, f_k^p\rangle\}$ where the set $\{a_1, \ldots a_k\}$ contains all abstract addresses that may be stored in x_p and where x_p itself holds the offset or value of `p`. Suppose the variable x_p corresponds to the ith dimension of the polyhedron $P \in \mathit{Poly}$ and $\langle f_1^p, \ldots f_k^p \rangle$ correspond to the next k dimensions, then the value of `p` is defined as follows:

$$\begin{aligned}\gamma_{\mathtt{p}}(P) := \{ v + f_1^p p_1 + \ldots f_k^p p_k \mid \; & \langle \ldots x_{i-1}, v, f_1^p, \ldots f_k^p, x_{i+k+1}, \ldots \rangle \in P \cap \mathbb{Z}^n \\ & \wedge \{ \langle a_1, f_1^p \rangle, \ldots \langle a_k, f_k^p \rangle \} = A(x_p) \\ & \wedge p_i \in \rho(a_i), i = 1 \ldots k \}\end{aligned}$$

The above concretisation interprets the Boolean variables in the domain as multiplier for the addresses with which they are associated with. Thus, a `NULL` pointer is characterised by all flags being zero in the polyhedron and an explicit NULL tag in $\mathcal{A}$ is not necessary anymore. The above interpretation simplifies the evaluation of linear expressions: Consider calculating `e=p+q-r` and suppose that $A(x_p) = \{ \langle a_1, f_1^p \rangle, \ldots \langle a_k, f_k^p \rangle \}$ and similarly for x_q, x_r and x_e. Then the result is calculated by updating the polyhedral variables $x_e, f_1^e, \ldots f_k^e$ of `e` as follows:

$$\begin{aligned} x_e &= x_p + x_q - x_r \\ f_1^e &= f_1^p + f_1^q - f_1^r \\ &\;\;\vdots \\ f_k^e &= f_k^p + f_k^q - f_k^r \end{aligned}$$

Thus, rather than matching certain common patterns for pointer subtraction and pointer offset expressions, the Boolean flags can be added component-wise. Since this makes it possible to add several Boolean flags together, the value of a particular f_i^e might not be in $[0, 1]$. As a consequence, an access through the pointer `e` with offset x_e in state P has to be checked as follows:

- $P \sqsubseteq [\![\{ 0 \le f_1^e \le 1, \ldots 0 \le f_k^e \le 1 \}]\!]$: `e` contains each l-value at most once.
- $P \sqsubseteq [\![f_1^e + \ldots + f_k^e \le 1]\!]$: `e` contains at most one l-value.
- $P \sqsubseteq [\![f_1^e + \ldots + f_k^e \ge 1]\!]$: `e` is not `NULL`.

The semantic function for the pointer access is then executed k times, once for each target a_i and state $P \sqcap [\![\{ f_i^e = 1 \}]\!]$ where $i = 1, \ldots k$.

Evaluating a conditional $a \; op \; b$ where $op \in \{ <, \le, =, \ne, \ge, > \}$ has to take the points-to sets of the expressions a and b into account. Suppose that the points-to sets of a and b are given by $A(x_a) = \{ \langle a_1, f_1^a \rangle, \ldots, \langle a_k, f_k^a \rangle \}$ and $A(x_b) = \{ \langle a_1, f_1^b \rangle, \ldots, \langle a_k, f_k^b \rangle \}$ and that no more than one flag is set in each expression. The semantics of $a \; op \; b$ in the state P can then be calculated by considering different combinations of flags $f_1^a, \ldots f_k^a$ and flags $f_1^b, \ldots f_k^b$. For instance, for all states $P' = P \sqcap [\![\{ f_1^a = f_1^b = 0, \ldots f_{i-1}^a = f_{i-1}^b = 0, f_i^a = f_i^b = 1, f_{i+1}^a = f_{i+1}^b = 0, \ldots f_k^a = f_k^b = 0 \}]\!]$, the comparison can be evaluated as $P' \sqcap [\![a \; op \; b]\!]$ since the addresses that are implicitly present in each side of the condition are the same (unless $|\rho(a_i)| > 1$, which indicates that a_i summarises several addresses). All combinations in which the flag set for a is different to that from b indicate that different pointers are compared which can be flagged as erroneous. In case a pointer is compared with a value, the address contained in the pointer has to be made explicit by adding a range of possible addresses to the expression. Possible addresses of variables are typically $[4096, 2^{32} - 1]$ on a 32-bit machine, since no variable can be stored in the first page of memory. Thus, given the test `p==NULL` and a points-to set of `p` of $A(x_p) = \{\langle a, f\rangle\}$, the input state P can be transformed to $P_1 \sqcup P_2$ where $P_1 = P \sqcap [\![\{f = 1, x_p + [4096, 2^{32} - 1] = 0\}]\!]$ and $P_2 = P \sqcap [\![\{f = 0, x_p = 0\}]\!]$. Here, P_1 is empty if the pointer `p` has a zero offset, in which case $P_1 \sqcup P_2 \sqsubseteq [\![\{f = 0\}]\!]$, that is, `p` has an empty points-to set.

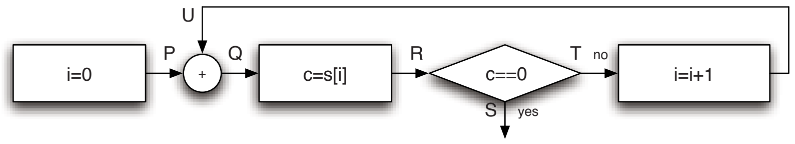
This concludes the presentation of the abstract semantics in the context of points-to sets that are guarded by Boolean flags. We now present a practical problem in which refined points-to information is key to a successful verification.

4.2 Boolean Flags and String Buffer Analysis

Consider the task of advancing a pointer `s` by executing the loop `while(*s) s++`. Suppose `s` points to a string buffer whose length is given by the first element with value 0. For ease of presentation, we augment and expand the loop as follows:

```
char s[11] = "the␣string";
int i = 0;
while (true) {
  c = s[i];
  if (c==0) break;
  i = i+1;
};
```

The task is to check that the string buffer `s` is only accessed within bounds. In order to simplify the presentation, we define the polyhedral operations and discuss the stability of the fixpoint. To this end, let $n \in \mathcal{X}$ represent the index of the first zero in `s`, i.e. the position of the ASCII character NUL. We decorate the control flow graph of the above loop with polyhedra P, Q, R, S, T, U as follows:



The initial values of the program variables is described by $P = [\![\{i = 0, n = 10\}]\!]$. The join of P and the back edge that is decorated by $U \in \mathit{Poly}$ is defined as $Q = P \sqcup U$. To verify that the array access `s[i]` is within bounds, we compute $Q' = Q \sqcap [\![\{0 \leq i \leq 10\}]\!]$ and issue a warning if $Q \not\sqsubseteq Q'$. We use the projection operator $\exists_{x_i}(P) = \{\langle x_1, \ldots x_{i-1}, v, x_{i+1}, \ldots x_n\rangle \mid \langle x_1, \ldots x_n\rangle \in P, v \in \mathbb{Q}\}$ to remove all information on x_i. The following definition of R is explained below:

$$\begin{aligned} R = {} & (\exists_c(Q) \sqcap \{i < n, 1 \leq c \leq 255\}) \\ & \sqcup (\exists_c(Q) \sqcap \{i = n, c = 0\}) \\ & \sqcup (\exists_c(Q) \sqcap \{i > n, 0 \leq c \leq 255\}) \end{aligned}$$

We assume that no information is tracked about the contents of the array, except for the NUL character at position n. The above definition of R therefore

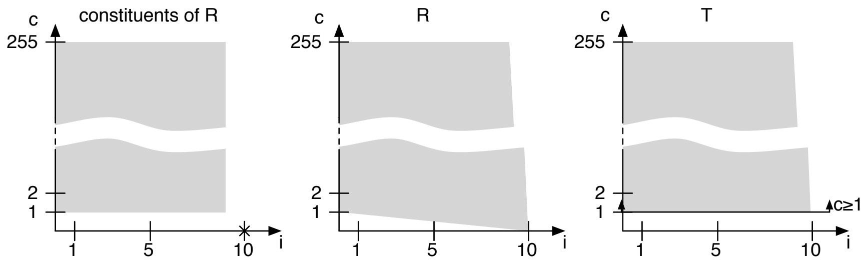


Fig. 6. The fixpoint of the state spaces after accessing the buffer

defines the read character c in terms of the value of the pointer offset i in Q and restricts c to the desired range. Specifically, the value of c is restricted to $[1, 255]$ if $i < n$, it is set to 0 if $i = n$ and to $[0, 255]$ if $i > n$. The last three equations that comprise the system are given by the following definitions:

$$S = R \sqcap [\![c = 0]\!]$$
$$T = (R \sqcap [\![c < 0]\!]) \sqcup (R \sqcap [\![c > 0]\!])$$
$$U = \{\langle x_1, \ldots i + 1, \ldots x_n\rangle \mid \langle x_1, \ldots i, \ldots x_n\rangle \in T\}$$

The test `c!=0` is implemented by joining the two state spaces $R \sqcap [\![c < 0]\!]$ and $R \sqcap [\![c > 0]\!]$. Note that $R \sqcap [\![c < 0]\!] = \emptyset$ as the definition of R confines c to the range $[0, 255]$. Figure 6 shows the state space R at the fixpoint. Specifically, the first graph shows the contribution of the first and second equation, the second graph shows the join of both states. The third graph shows the calculation of T in that R is intersected with $c > 0 \equiv c \geq 1$. The intersection reduces the upper bound of i to less than 10 and integral tightening ensures that $i \leq 9$ before i is incremented by one to define U. Since $U \sqsubseteq Q$, a fixpoint has been reached.

Observe that the character c acts as a Boolean flag, even though in the non-zero state c takes on many values. Specifically, c distinguishes the state where $i < n$ and $i = n$. Furthermore, all equations that act as guards in the definition of R have coefficients in $\{1, -1\}$. Thus, this Boolean behaviour can be exploited by splitting the control flow according to the value of c which makes it possible to prove that the access `s[i]` is within bounds using the weaker Octagon domain.

4.3 Accessing Several String Buffers

The last section demonstrated that, within the domain of polyhedra, an invariant such as $p < n$ (the access position lies in front of NUL position) can be recovered through the relational information in the domain merely by intersection with the loop invariant $c \neq 0$ (the read character is not NUL) which acts like a Boolean flag. This section demonstrates how string buffer analysis suffers from precision loss when no relational information exists between the polyhedral domain and the points-to domain. In particular, the following example demonstrates that the fixpoint is missed if the pointer may point to two different strings.

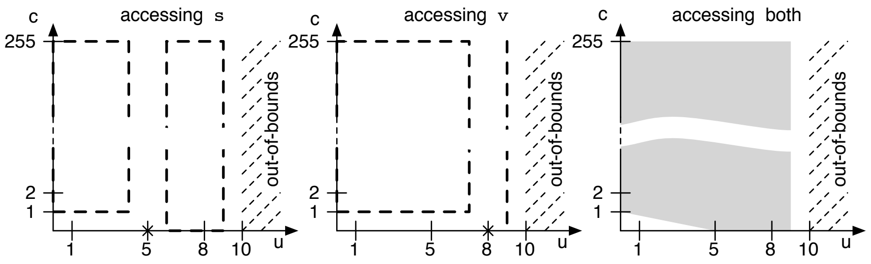


Fig. 7. Finding the NUL position in more than one buffer at a time is impossible without stating that a points-to set changes from a certain iteration onwards

```
char s[10] = "Spain";
char v[10] = "Valencia";
char* u = v;
if (rand()) u = s;
char *p = u;
while (*u) u++;
printf("length␣is␣%i\n", (u-p));
```

The shown code sets the pointer `u` to either point to the string `"Spain"` or to `"Valencia"`, depending on a random number. A backup of `u` is stored in `p` before `u` is advanced to the NUL position in the string buffer. The last statement prints the length of the accessed string by calculating the pointer difference `u-p`.

Let $u \in \mathcal{X}$ denote the offset of `u` and let a_v and a_s denote the abstract addresses of the variables `v` and `s`, respectively. Thus, when using a normal flow-sensitive points-to analysis, the points-to set of `u` can be described as $A(u) = \{a_v, a_s\}$. In this case, analysing the read access `*u` in the loop condition has to assume that both string buffers might be accessed which can be implemented by evaluating the semantic equation of R in Sect. 4.2 twice, once with the NUL position of `s` and once with that of `v` and joining the result. Let $c \in \mathcal{X}$ denotes the value of the read character. The value of c with respect to the access position u is shown in Fig. 7. While the analysis can infer that c may be zero as soon as $u \geq 5$ (from the first graph), the analysis correctly infers that the loop may continue to increment u since the NUL position in `v` at index 8 has not been reached yet. However, once this index is reached, the value of c resulting from accessing `s` can either be zero or non-zero, as the access lies past the first known NUL position. Hence, $c > 0$ and the analysis assumes that the loop can continue to iterate, leading to the state in the third graph. In the next iteration the analysis will emit a waring that the loop will access both buffers past their bound.

The example can readily be verified using the revised points-to abstraction presented in Sect. 4.1 such that $A(u) = \{\langle a_s, f_s\rangle, \langle a_v, f_v\rangle\}$. The statements before the loop ensure that $0 \leq f_v \leq 1, f_s = -f_v$. The semantics of the access `*u` is calculated by evaluating the equation for R firstly with $Q \sqcap [\![\{f_s = 1\}]\!]$ and `s` and secondly with $Q \sqcap [\![\{f_v = 1\}]\!]$ and `v`. Analysing the loop twice, once with $f_s = 1$ and once with $f_v = 1$ will implicitly restrict the access to one of the string buffers, thereby inferring that the loop terminates with $u = 5$ and $u = 8$, respectively. Alternatively, the loop may be analysed once. In this case the analysis infers a linear relationship between the pointer offset and f_v in that $u \geq 6$ implies $f_v = 1$ and $f_s = 0$. Thus, in the next evaluations of the loop, calculating $Q \sqcap [\![f_s = 1]\!]$ before evaluating the access to the buffer a_s will restrict u to values smaller than 5. Hence, termination of the loop is inferred when $u = 5, f_s = 1$ or $u = 8, f_v = 1$: The states are $S = [\![\{c = 0, 3f - u = 5, u \geq 5\}]\!]$ and $R = [\![\{c+255u-765f_v \leq 1275, c \leq 255, 5c+u \geq 5, 8c+u-3f_v \geq 5, u \geq 0\}]\!]$.

5 Discussion and Conclusion

The last section demonstrated how Boolean flags help to refine the abstract semantics of pointer accesses, yielding a more precise abstraction. From Sect. 3 it follows that adding a Boolean flag is equivalent to analysing those parts of a program twice for which the Boolean flag can take on both values. Thus, the natural question to ask is which approach is cheaper. In order to get an initial answer to this question, we implemented the semantics of the above loop using both techniques. As numeric domain, we used general polyhedra [20] as the loop invariant requires three variables and is therefore beyond the reach of the TVPI domain [22]. Interestingly, inferring the given invariant does not require any integral tightening techniques as the three guards $i < n$, $i = n$ and $i > n$ in the semantic equation defining R discard rational solutions since $i < n$ merely abbreviates $i \leq n - 1$. For comparable results, neither widening nor any approximation of the convex hull was applied. Since the analysis of the loop does not take very long, we calculated the fixpoint one million times in order to get more accurate results. Nevertheless, any variation in the implementation can have an immense effect on the given numbers, such that they have to be taken with a grain of salt. Analysing the loop twice, once for s, once for v, requires $7.7\mu s$ per two fixpoint calculations. Analysing the loop once using a Boolean flag requires $5.6\mu s$ for one fixpoint calculation. Interestingly, if both strings are of length 8, the analysis speeds up to $4.8\mu s$ as the state contains more equalities that can be factored out. While our experiment suggests that using flags is preferable, their exponential cost may grind an analyser to a halt. For example, using the character c to distinguish states quickly leads to inequalities with very large coefficients. Integral tightening as implementable in the TVPI domain could help to reduce the size of coefficients. However, the TVPI domain is too weak to verify the example above. One way of making TVPI polyhedra more expressive is to track linear inequalities over more than two variables symbolically. Such inequalities would take on the role of propagation rules in the context of Constraint Handling Rules [9]. Reconsider the first example on string buffers where $c = 0$ implies that $p < n$. In case n is not constant, but may take on a range of values, the termination of the loop in Sect. 4.2 cannot be proved since three variables are necessary to express the invariant [18]. Instead of re-analyzing the code for $1 \leq c \leq 255$ and $c = 0$ it is possible to track the implication $c = 0 \Rightarrow p < n$ and automatically insert the inequality $p < n$ whenever the interval of c is restricted to $[0, 0]$. Future work will asses if this trick can also be applied to Boolean flags.

In conclusion, we proposed to use Boolean flags to separate states in a single convex polyhedron as an alternative to partitioning the set of traces by re-analysing code. We demonstrated their use in applications such as points-to analysis where Boolean flags can be superior to analysing code twice.

References

1. Bagnara, R., Hill, P.M., Zaffanella, E.: Not Necessarily Closed Convex Polyhedra and the Double Description Method. FAC 17(2), 222–257 (2005)
2. Bagnara, R., Hill, P.M., Zaffanella, E.: An Improved Tight Closure Algorithm for Integer Octagonal Constraints (2008)
3. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S., Ustuner, A.: Thorough Static Analysis of Device Drivers. In: European Systems Conference, pp. 73–85. ACM, New York (2006)
4. Ball, T., Rajamani, S.K.: Bebop: A Symbolic Model Checker for Boolean Programs. In: SPIN Workshop on Model Checking and Software Verification, London, UK, pp. 113–130. Springer, Heidelberg (2000)
5. Ball, T., Rajamani, S.K.: Automatically Validating Temporal Safety Properties of Interfaces. In: SPIN Workshop on Model Checking of Software, New York, NY, USA, pp. 103–122. Springer, Heidelberg (2001)
6. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Principles of Programming Languages, pp. 269–282 (1979)
7. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE Analyzer. In: European Symposium on Programming, Edinburgh, Scotland, pp. 21–30. Springer, Heidelberg (2005)
8. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Constraints among Variables of a Program. In: Principles of Programming Languages, Tucson, Arizona, pp. 84–97. ACM Press, New York (1978)
9. Frühwirth, T.: Theory and Practice of Constraint Handling Rules. Journal of Logic Programming, Special Issue on Constraint Logic Programming 37(1-3), 95–138 (1998)
10. Gopan, D., Reps, T.W.: Low-Level Library Analysis and Summarization. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 68–81. Springer, Heidelberg (2007)
11. Harvey, W.: Computing Two-Dimensional Integer Hulls. SIAM Journal on Computing 28(6), 2285–2299 (1999)
12. Heintze, N., Tardieu, O.: Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In: Programming Language Design and Implementation, pp. 254–263 (2001)
13. Lagarias, J.C.: The Computational Complexity of Simultaneous Diophantine Approximation Problems. SIAM Journal on Computing 14(1), 196–209 (1985)
14. Mauborgne, L., Rival, X.: Trace Partitioning in Abstract Interpretation Based Static Analyzers. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 5–20. Springer, Heidelberg (2005)
15. Miné, A.: The Octagon Abstract Domain. Higher-Order and Symbolic Computation 19, 31–100 (2006)

16. Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM 8, 102–114 (1992)
17. Schrijver, A.: Theory of Linear and Integer Programming. John Wiley & Sons (1998)
18. Simon, A.: Value-Range Analysis of C Programs. Springer (to appear, 2008)
19. Simon, A., King, A.: Analyzing String Buffers in C. In: Kirchner, H., Ringeissen, C. (eds.) AMAST 2002. LNCS, vol. 2422, pp. 365–379. Springer, Heidelberg (2002)
20. Simon, A., King, A.: Exploiting Sparsity in Polyhedral Analysis. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 336–351. Springer, Heidelberg (2005)
21. Simon, A., King, A.: Taming the Wrapping of Integer Arithmetic. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 121–136. Springer, Heidelberg (2007)
22. Simon, A., King, A., Howe, J.M.: Two Variables per Linear Inequality as an Abstract Domain. In: Leuschel, M. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 71–89. Springer, Heidelberg (2003)

Reasoning about Control Flow in the Presence of Transient Faults[⋆]

Frances Perry and David Walker

Princeton University
{frances,dpw}@cs.princeton.edu

Abstract. A transient fault is a temporary, one-time event that causes a change in state or erroneous signal transfer in a digital circuit. These faults do not cause permanent damage, but when they strike conventional processors, they may result in incorrect program execution. While detecting and correcting faults in first-order data may be accomplished relatively easily by adding redundancy, protecting against faults during control flow transfers is substantially more difficult. This paper analyzes the problem of maintaining the control-flow integrity of a program in the face of transient faults from a formal theoretical perspective. More specifically, we augment the operational semantics of an idealized assembly language with additional rules that model erroneous control-flow transfers. Next, we explain a strategy for detecting control-flow errors based on previous work by Oh [10] and Reis [15]. In order to reason about the correctness of the strategy relative to our fault model, we develop a new assembly-level type system designed to guarantee that any control flow transfer to an incorrect block will be caught before control leaves that block. The key technical result of the paper is a rigorous proof of this fundamental control-flow property for well-typed programs.

1 Introduction

In recent decades, microprocessor performance has been increasing exponentially, due in large part to smaller and faster transistors. While such transistors yield performance enhancements, their lower threshold voltages and tighter noise margins make them less reliable [3,9,17], rendering processors that use them more susceptible to *transient faults*. These faults do not cause permanent damage, but may result in incorrect program execution by altering signal transfers or stored values. While transient faults are currently rare, they have caused significant failures in server farms at companies including AOL and eBay [4] and in supercomputers such as those at Los Alamos Labs [8]. More importantly, current hardware manufacturing trends suggest the problem of transient faults will grow substantially in the future [6].

[⋆] This research is funded in part by NSF award CNS-0627650 and a Microsoft graduate fellowship. We would like to thanks Andrew Appel, David August, George Reis and Neil Vachharajani for many enlightening discussions on transient faults, harware mechanisms and fault tolerance.

© Springer-Verlag Berlin Heidelberg 2008

In order to counter the threat of transient faults, researchers from industry and academia have been searching for solutions to the reliability problem in both hardware and software. Broadly speaking, with sufficient hardware resources, hardware-only solutions are more efficient for a single, fixed reliability policy, but software-only solutions are more flexible and less costly. In terms of flexibility, software-only solutions may be deployed *immediately* on current hardware that already exists in the field, simply by recompiling the application in question. In terms of cost-effectiveness, recent studies have shown that software techniques for fault tolerance often add approximately 35% overhead [15] to the computation with no additional hardware cost, whereas a standard double- or triple- modular redundancy technique will add 100% or 200% to the hardware cost, with some additional performance overhead for communication between replicas. Hence, depending upon where a given application sits in the cost-performance-reliability trade-off space, software, hardware or some mix of the two may be the preferred solution.

Unfortunately, devising software solutions to the problem of transient faults, and making sure they are correct, is an extremely difficult task. Just as the many possible interleavings of threads make it difficult to reason about the properties of concurrent programs, the many possible scenarios in which transient faults can arise make it difficult to reason about the properties of faulty programs. Moreover, just as conventional testing is often an ineffective way to uncover bugs in concurrent programs, testing is likely to be an ineffective way to uncover reliability errors in possibly faulty programs.

Faced with these challenges, we and other researchers at Princeton have recently begun to develop type-theoretic techniques for reasoning about software in the presence of transient faults. In our first effort [18], we devised a lambda calculus called λ_{zap} to serve as a highly idealized model for unreliable computations. The operational semantics of the calculus specify that any value may suddenly be corrupted during execution. However, programs are able to replicate computations and use atomic voting operations to check replicas against one another to detect and recover from transient faults. A type system for λ_{zap} guarantees that any well-typed program is fault tolerant. In our second piece of work [11], we studied fault tolerance in the more realistic setting of assembly language with specialized hardware instructions to aid detection of faults. Once again we devised a type system (this time called TAL_{FT}) and rigorously proved that it guarantees a strong fault tolerance property for all well-typed programs. From a theoretical perspective, these type systems codify formal reasoning techniques that allow programmers to prove strong reliability properties of their programs. Equally importantly, from a practical perspective, these type systems can be implemented and used to check the correctness of compiler outputs. Using a type checker to verify these reliability properties, where possible, is vastly superior to conventional testing as the type checker gives *perfect coverage* relative to the fault model whereas any test suite will be highly incomplete.

Despite the progress made to date, this prior work skirts the issue of how to reason about code that not only incurs faults to first-order data, but also may go wrong during a control flow transfer. The faulty lambda calculus λ_{zap} avoids the issue altogether by assuming the existence of high-level atomic operations to simultaneously check for errors, recover and jump to a new control flow point. The fault-tolerant typed assembly language TAL_{FT} admits the possibility of faults to the program counter, but requires a highly specialized instruction set and additional hardware state to detect those faults.

Surprisingly, however, researchers [10,15,5] have developed techniques for detecting certain classes of control-flow errors entirely in software. These techniques do not catch all control-flow errors, but empirical evidence suggests they can improve system reliability substantially. However, many theoretical questions remain. In particular, is it possible to characterize the effectiveness of these techniques *analytically* as opposed empirically? In other words, can we prove that such techniques are sound with respect to an interesting and non-trivial, though incomplete, fault model? One of the key benefits of such an analysis is that it would guarantee an important fragment of the problem has been thoroughly solved and thereby free researchers to study auxiliary instrumentation techniques that address the remaining incompleteness. Perhaps more importantly, the formal fault model would define an important hardware/software interface: The software has been proven to handle faults that lie within the model; future hardware designers need only provide mechanisms to catch those faults that lie outside the model. While this latter point may appear of little importance since TAL_{FT} already demonstrates how to design a sound hybrid hardware-software protection system, the key difference is that such results would show how to shift a substantial portion of the control-flow checking burden from the hardware to software. This may lead to much simpler hardware designs as well as the opportunity to trade performance for reliability at *compile time* as opposed to *hardware design time*.

In this paper we attack these theoretical questions following a similar strategy to our earlier work. First, we define an incomplete, yet simple, elegant and non-trivial control-flow fault model — one in which faults can cause jump instructions and conditional branches to transfer control to the beginning of any program block. Next, we develop a type system that guarantees a strong fault tolerance property relative to this fault model. We have proven our type system is sound and also have demonstrated that it is sufficiently expressive that we can compile classic while programs into well-typed programs in the language. Due to space limitations, this extended abstract only describes selected elements of our assembly language and its type system. Further details may be found in an accompanying technical report [12] and in our online proofs [13].

2 Informal Overview

When a transient fault causes the actual sequence of control flow blocks visited by a program to deviate from the expected sequence, we say a control-flow error has occurred. In this paper, control-flow errors arise when a fault effects either (1) the target address of a jump instruction, (2) the target address of a conditional jump instruction, or (3) the boolean condition of a conditional jump instruction. Such faults may occur immediately prior to attempting the control-flow transfer or at any other time during the computation. However, whenever a control-flow operation is executed, we assume control is either transferred to the beginning of some block or to an illegal instruction, which is immediately detected by the hardware. We currently do not consider the possibility that a fault causes a control flow transfer to a legal instruction in the middle of a block. (For a discussion on alleviating this restriction, see Section 6.) In addition, we adhere to the standard *Single Event Upset* model [14,16], which states that only one fault may occur during an execution, though faulty values may be copied, propagated and used in any way an ordinary value may be used.

In order to ensure that control flow transfers do not go wrong, compiled code computes a replica of the intended control-flow destination prior to the control-flow transfer and moves the intended destination into a designated register. We refer to this register as the *intentions register* `ri`. This intentions register is part of the global "calling convention" for fault-tolerant control flow transfers. We fix the register so that all jump targets know where to find the intended destination, even when there has been a control-flow fault.

As an example, to jump to address `L2`, one might use the following code sequence. In this code, we leave ellipsis in between instructions to emphasize our system allows flexible scheduling of instructions — ordinary instructions may be interleaved with the instructions used to guarantee fault tolerance.

```
L1: ...; movi ri, L2; ...; movi r2, L2; ...; jmp r2
```

Since the intentions register `ri` plays a special role in the protocol for detecting control-flow errors, we will need to type check the move instruction that loads this register in a special way. To designate the move as special, we henceforth write it `intend L2` rather than `movi ri, L2` as in the following example code.

```
L1: ...; intend L2; ...; movi r2, L2; ...; jmp r2
```

If the intentions register has been set properly prior to all jump instructions, the jump targets are able to catch control flow errors. To be specific, all jump targets should be instrumented with the following code.

```
Lk: movi r2, Lk; ...; sub r2, r2, ri; ...; brnz r2, Lrecover; ...
```

Here, the current block address `Lk` is loaded into register `r2` and then compared with the contents of `ri` and if there is any difference, control is transferred to `Lrecover`, an address containing recovery code.[1] Once again, since the branch to the recovery code plays a special role in the fault-tolerance protocol, we give it the special syntax `recovernz r2`. Thus, our detection code will henceforth be written as follows.

```
L2: movi r2, L2; ...; sub r2, r2, ri; ...; recovernz r2; ...
```

As an example of how a transient fault might be caught using our protocol, suppose register `r2` is corrupted just prior to attempting to execute the jump to

[1] Since recovery is a secondary issue to detection, we do not consider it in this paper.

L2 in block L1. If the corrupted value is not a valid code address, then a hardware fault will be triggered. Otherwise, upon arrival at some erroneous control flow block, say L3, the intended destination L2 remains safely untouched in register ri, though, unnervingly, all other program invariants may be disrupted. The target code compares the contents of ri (*i.e.,* L2) with L3, which it loaded into r2 after arriving at the current block. It detects a difference and jumps to the recovery code.

One must also consider what happens if faults strike at different times or in different places. If ri is corrupted, it appears as though there was a fault because ri differs from the current block label (assuming the fault occurs prior to the subtraction). Unable to tell the difference between a fault in the intentions register and a fault in the control-flow transfer itself, we jump to recovery code. A number of other scenarios must also be analyzed — in order to have confidence in the solution, one must do so in a principled, disciplined fashion. It is important to observe that similar, but subtly different code sequences do not adequately protect against faults. In particular, optimizations like copy propagation, common subexpression elimination and some code motion transformations, are no longer semantics-preserving in the context of transient faults.

For example, the code motion transformation illustrated below shifts the move from a target block into the jumping block and creates a vulnerability.

```
L1: ...; movi r2, L2; intend L2; movi r3, L2; jmp r2
Lk: sub r3, r3, ri; recovernz r3; ...
```

Above, a fault to r2 causes a control-flow error, but testing r3 against ri at the recovernz instruction will not help detect the fault.

The protocol for handling conditional branches is slightly more involved than the case for jumps, but follows a similar pattern. We begin by assuming that the the jump target is held in registers r3 and r3' and the condition for the jump is held in registers r4 and r4'. These register pairs must be *independent replicas* of one another. In other words, in the absence of faults, they should contain the same value, and moreover, a fault to one should have no impact on the value of the other. Given this assumption (which will be verified by our type system), the following code sequence sets up a conditional branch, which may fall through to L2 or may jump to the target in r3. It also uses a conditional move cmovz r4', ri, r3', which moves the contents of r3' into ri if r4' is zero, and otherwise does nothing.[2]

```
L1: ...; intend L2; cmovz r4', ri, r3'; brz r4, r3
```

Again, to notate the special role of ri and simplify the presentation, we will henceforth write the conditional move cmovz r4', ri, r3' as intendz r4', r3'. Intuitively, the intend instruction unconditionally sets the intentions register, whereas the intendz instruction conditionally sets the intentions register.

[2] Many architectures including the IA-32 following the Pentium Pro, the Sparc V-9 and the IA-64 have conditional moves. Other architectures can use a conditional branch and a move instruction instead, but this branch will not be protected.

colors	$c ::= G \mid B \mid O$	*instructions* i	$::=$ `movi` $r_d\ v$
colored values v	$::= c\ n$		$\mid$ `sub` $r_d\ r_s\ r_s$
			$\mid$ `intend` r_t
code memory	$C ::= \cdot \mid C, \ell \to b$		$\mid$ `intendz` $r_z\ r_t$
			$\mid$ `recovernz` r_z
registers	$r ::= r_{¡} \mid r_1 \mid \ldots \mid r_n$	*blocks*	$b ::= i; b \mid$ `jmp` $r_t \mid$ `brz` $r_z\ r_t$
register file	$R ::= \cdot \mid R, r \to v$		
		states	$\Sigma ::= (C, h, R, b)$
history	$h ::= \ell_1, \ldots, \ell_n$	*final states*	$\mathcal{F} ::= \Sigma \mid$ `recover`$(h) \mid$ `hwerror`(h)

Fig. 1. Machine State Syntax

Summary. By creating duplicate copies of intended control flow targets, it is possible to check that control has arrived at the proper location. In the following sections, we make the machine's operational semantics and fault model precise and develop a sound type system strong enough to verify that the "good" instruction sequences we have discussed in this section are indeed fault tolerant.

3 The Control-Flow Machine

For clarity and elegance, we will work with a minimal assembly instruction set involving move (`movi`), subtraction (`sub`), jump (`jmp`) and conditional branch if zero (`brz`) instructions as well as the special macros `intend`, `intendz` and `recovernz`. Instruction operands include constant values v and registers r. In the previous section, values were unannotated, but from this point forward we annotate every value with a *color* c where c is either G (green), B (blue) or O (orange). These colors have no operational significance, but they play a special role in the type system and proof of correctness. The only kind of value is an integer. In general, meta-variable n ranges over integers, but we use meta-variable ℓ to emphasize that an integer will be used as an address.

Instructions are grouped together in code blocks b that are always terminated by either a jump or a conditional branch instruction. Code memory C is a partial map from addresses to valid code blocks b. Addresses are ordered, and the notation $\ell + 1$ refers to the address of the block following the block at ℓ. If a block at ℓ ends with a conditional branch, $\ell + 1$ must inhabit the domain of C.

The register file R is a mapping from registers to the colored values they contain. The registers include the intentions register $r_{¡}$ and a number of general-purpose registers r_1 through r_n. We use the notation $R(r)$ to denote the contents of r in R. We use the notation $R[r \mapsto v]$ to denote a new register file R' created by updating R so it maps r to v. When we wish to refer to the unannotated integer n as opposed to the colored value $c\ n$ in a register r in R, we use the notation $R_{val}(r)$. Similarly, $R_{col}(r)$ refers to the color annotating the value in r.

An ordinary abstract machine state Σ is a tuple containing code C, history h, register file R and code block to be executed b. The history h is a sequence of labels. It records the code blocks visited during the current execution. In

Static Expressions		Types	
exp kinds	$\kappa ::= \kappa_{int} \mid \kappa_{hist}$	*stage description*	$\rho ::= check \mid ok$
exp contexts	$\Delta ::= \cdot \mid \Delta, x : \kappa$		$\mid go \mid goz$
exps	$e ::= x \mid n \mid e - e$	*basic types*	$\tau ::= int \mid \rho \mid \forall[\Delta](\Gamma, \sigma)$
	$\mid e?e : e$	*value types*	$t ::= \langle c, \tau, e \rangle$
substitutions	$S ::= \cdot \mid S, e/x$	*type option*	$\tau\ opt ::= \tau \mid \text{undef}$

Context Typing		ZapTags	
heap typing	$\Psi ::= \cdot \mid \Psi, \ell \rightarrow \tau$	*zap tag*	$Z ::= \cdot \mid c \mid CF$
reg file types	$\Gamma ::= \cdot \mid \Gamma, r \rightarrow t$		
history typing	$\sigma ::= \epsilon \mid x \mid \sigma \circ e$		

Fig. 2. Typing Syntax

addition to ordinary abstract machine states, "final states" $\mathcal{F}$ include two special states. The state `recover`(h) represents a state in which a transient fault has occurred and has been caught. The labels in history h were visited during the execution. The state `hwerror`(h) represents a state in which a transient fault causes transition to an invalid address. Figure 1 summarizes the syntax of the assembly language and machine states.

3.1 Dynamic Semantics

We model the dynamic semantics of the assembly language using a small step operational semantics. In general, the single step operational judgments have the form $\Sigma \longrightarrow_k \mathcal{F}$ where k, which is either zero or one, records the number of faults that occur during the step.

The most interesting rules in the system are the rules modeling faults. The primary rule (*zap-reg*) arbitrarily corrupts the value in a single register, though the color tag (which has no operational significance) remains unchanged.

$$\frac{R(r) = c\ n}{(C, h, R, b) \longrightarrow_1 (C, h, R[r \mapsto c\ n'], b)}\ (\textit{zap-reg})$$

The rule above may fire at any time, just as a transient fault may occur at any point. In particular, it may fire just prior to execution of a jump (`jmp` r_t) or a branch (`brz` r_z r_t), corrupting the jump target in register r_t.

For uniformity in our fault model, we also consider errors in execution of the `recovernz` r_z instruction. These rules, as well at the rules for normal instruction execution, are provided in the technical report [12].

4 Typing

The overall design of the type system is based on two nonstandard concepts: (1) Classifying the reliability properties of values, and (2) Using abstract types to make sure that the fault tolerance protocol proceeds in the correct order, with no steps omitted or inappropriate steps inserted. The following paragraphs explain the main intuitions behind each concept.

Classifying the Reliability Properties of Values. Since faults occur completely unpredictably and at run time, it is not possible for the type system to know which values have incurred faults or to track the propagation of presumed faulty values precisely. Consequently, as is usual, the type system will have to approximate these properties somehow. It does so by assigning each value a color and ensuring that values with the same color have related reliability properties.

Most values either belong to the green group or to the blue group. These two groups have the property that they are *independent* and *redundant.* In other words, a fault in a green value can never percolate to a blue value and vice versa. Consequently, when corresponding green and blue values are compared, at least one of them must be correct, even when a fault has occurred. This mutual independence property is ensured by a series of simple checks in the type system that guarantee that green values are not used to construct blue values and vice versa.

But what if a control-flow fault *has* occurred? In that case, almost all program invariants are invalidated, including any properties of either blue or green values. However, *orange* values are manipulated in such a way as to preserve their properties in just this situation.

There are two general mechanisms by which one can guarantee orange values maintain their expected properties in the face of a control-flow fault. The first mechanism is to ensure that the orange value in question is not live across the control-flow transfer: If the value has been constructed in the current block and does not depend upon values in previous blocks, a control-flow error will not influence its properties. The second mechanism involves ensuring that every possible control-flow transfer maintains the invariant in question. If the invariant is true across *every* control-flow transfer, then it is true no matter where control winds up. This second mechanism is used to classify the contents of r_{i} as orange across every control-flow transfer. Just as the type system isolates green values from blue and blue from green, orange is also isolated from the other two. Again, the purpose is to avoid having a fault in one color influence the others.

While values are classified using colors, entire machine states are classified using a related concept called *zap tags.* Intuitively, each zap tag specifies which colors may no longer be trusted. For example, if zap tag Z is empty (written "$\cdot$"), then there have been no faults during the computation, and all values, no matter what their color, satisfy the standard invariants associated with their compile-time type. On the other hand, if Z is a color c, then values with color c may have been corrupted, but other values will be correct. The final zap tag CF classifies machine states after a control-flow error has occurred. In this case, we know nothing about green or blue values, but the properties of orange values remain valid. The table below summarizes the properties that hold under each zap tag while in block ℓ. A value is *trusted* if it satisfies standard canonical forms properties (*e.g.,* a value with code type is actually a pointer to valid code). The table says a value is *untrusted* when the standard canonical forms properties do not necessarily hold.

Zap Tag	G **values**	B **values**	O **values**	ℓ **is the intended destination**
$\cdot$	trusted	trusted	trusted	yes
G	*untrusted*	trusted	trusted	yes
B	trusted	*untrusted*	trusted	yes
O	trusted	trusted	*untrusted*	yes
CF	*untrusted*	*untrusted*	trusted	*no*

A zap tag Z is a subtype of another Z', written $Z \leq Z'$, when the values in machine states classified by Z are more trusted than the values in machine states classified by Z'. Hence the empty zap tag is a subtype of all other zap tags, and both B and G zap tags are a subtype of CF.

Typing Protocol Stages. The instructions in each block can be thought of as being divided into three distinct stages – the *checking code*, the *block body*, and the *exit code*. Each of these stages has its own distinct invariants. The type of intentions register r_{i} encodes the current stage and ensures that the stages occur in the correct order. It also guarantees no part of the protocol can be omitted or any inappropriate instruction added. These stages may be summarized as follows.

1. The checking code compares the intended target with the current location to determine if there has been a control flow fault.
2. In the block body, we already know the control flow correctly transferred to this block. At the end of this sequence, there is some green register that holds the target label for the next control flow transfer and some blue register that holds the duplicate copy of this label. In the absence of faults, these two values are equal.
3. The exit code sequence sets the intended target and transfers control to the new block.

The following subsections elucidate some of the technical ideas behind these intuitions. The complete definitions are described more thoroughly in our technical report [12].

4.1 Value Typing

The type of a value is a triple $\langle c, \tau, e\rangle$. The color c is assigned according to the intuitions expressed in the previous subsection. The *basic type* τ is either an integer type (*int*), a code type ($\forall[\Delta](\Gamma, \sigma)$), or a special type ρ that indicates the state of the fault tolerance protocol. The *static expression* e describes the value in more detail. These static expressions are used by the expression typing rules to require that blue and green computations compute identical results in the absence of faults. The expressions include variables x, integers n, subtraction $e_1 - e_2$ and conditional expressions $e_1 ? e_2 : e_3$ which equal e_2 when e_1 is non-zero and e_3 when e_1 is zero. The judgment $\Delta \vdash e : \kappa$ holds when all free variables in e are contained in the context Δ. The judgments $\Delta \vdash e_1 = e_2$ and $\Delta \vdash e_1 \neq e_2$

hold when the relation holds for all substitutions of the variables in Δ. The judgment $\Delta \vdash S : \Delta'$ holds when S provides substitutions for all variables in Δ', and the substituted expressions are well-formed in Δ.

The value typing judgment has the form $\Delta; \Psi \vdash^{Z} v : t$. Here, Δ contains expression variables free in t and the heap type Ψ maps integer addresses to basic types. The zap tag Z characterizes the current state of the machine as explained earlier. Z is always the empty tag when a user checks a program at compile time. It only takes on other values at run time for the purposes of the proof of preservation.

The central rule expresses the fact that if a value n has basic type τ, is equal to e and annotated with color c then it can always be given the type $\langle c, \tau, e\rangle$. However, if the zap tag Z is a color c, then all values $c\ n$ can also be typed using any basic type and any well-formed expression. Another key rule expresses the fact that when the zap tag is CF, green and blue values can be given *any* type. In particular green values may be given blue types and vice versa.

The type system also uses a subtyping judgment with the form $\Delta \vdash t \leq t'$. As an example, this judgment allows type $\langle c, \tau, e\rangle$ to be a subtype of $\langle c, int, e'\rangle$ whenever $\Delta \vdash e = e'$.

4.2 Instruction and Block Typing

Figure 3 presents several rules from the key judgments for checking program code. The first judgment has the form $\Delta; \Psi; \Gamma \vdash i : \Gamma'$. As before, Δ contains free expression variables and Ψ types heap addresses. Γ acts as the precondition for the instruction, mapping registers to types required prior to execution of the instruction. Γ' acts as the post condition for the instruction, mapping registers to types guaranteed after execution of the instruction.

The simplest instruction to type check is the `movi` $r_d\ c\ n$ instruction. It updates the type of the destination register r_d to be $\langle c, int, n\rangle$. The subtraction instruction `sub` $r_d\ r_a\ r_b$ requires that the values being subtracted are integers. Notice it also requires the integers arguments have the same color as the result – this restriction prevents faults in values with one color to influence another. Neither rule places any restrictions on the type of r_{i}, so they can occur during any stage of a block.

Though `intend` r_t is operationally the same as `movi` $r_{\mathsf{i}}\ r_t$, its typing rule requires that the intentions register r_{i} has basic type *ok*. This restriction guarantees any new intend will occur after the checking code has been completed. Notice also that the intention register is marked blue – in contrast, the address used as the real jump target will be marked green. Finally, the type of r_{i} is updated to reflect the new static expression and the new stage *go*.

A sequence of instructions is typed using the block typing judgment, which has the form $\Delta; \Psi; \Gamma; \sigma; e_i; \tau\ opt \vdash b$. In addition to Δ, Ψ, and Γ, the block typing judgment is parametrized by a sequence σ, an expression e_i, and a type option $\tau\ opt$. The sequence σ contains a list of expressions that describe the locations in the current history h. The expression e_i describes the intended target when the transfer occurred to the current label ℓ. If control flow correctly transferred

$$\boxed{\Delta; \Psi; \Gamma \vdash i : \Gamma'}$$

$$\frac{r_d \neq r_{\mathsf{i}}}{\Delta; \Psi; \Gamma \vdash \texttt{movi}\ r_d\ c\ n : \Gamma[r_d \mapsto \langle c, int, n\rangle]}\ (movi\text{-}t)$$

$$\frac{r_d \neq r_i \qquad \Gamma(r_a) = \langle c, int, e_a\rangle \qquad \Gamma(r_b) = \langle c, int, e_b\rangle}{\Delta; \Psi; \Gamma \vdash \texttt{sub}\ r_d\ r_a\ r_b : \Gamma[r_d \mapsto \langle c, int, e_a - e_b\rangle]}\ (sub\text{-}t)$$

$$\frac{\Gamma(r_i) = \langle c_i, ok, e_i\rangle \qquad \Gamma(r_t) = \langle B, \forall[\Delta_t](\Gamma_t, \sigma_t), e_t\rangle}{\Delta; \Psi; \Gamma \vdash \texttt{intend}\ r_t : \Gamma[r_{\mathsf{i}} \mapsto \langle B, go, e_t\rangle]}\ (intend\text{-}t)$$

$$\boxed{\Delta; \Psi; \Gamma; \sigma; e_i; \tau\ opt \vdash b}$$

$$\frac{\Delta; \Psi; \Gamma \vdash i : \Gamma' \qquad \Delta; \Psi; \Gamma'; \sigma; e_i; \tau\ opt \vdash b}{\Delta; \Psi; \Gamma; \sigma; e_i; \tau\ opt \vdash i; b}\ (sequence\text{-}t)$$

$$\frac{\begin{array}{l}\Gamma(r_{\mathsf{i}}) = \langle O, check, x_i\rangle \qquad \Gamma(r_z) = \langle O, int, e_z\rangle \qquad \Delta, x : \kappa_{int} \vdash e_z = e_\ell - x_i \\ \Delta \vdash \Gamma/r_{\mathsf{i}}/r_z \text{ wf} \qquad \Delta \vdash \sigma \text{ wf} \qquad \Delta \vdash e_\ell : \kappa_{int} \\ \Gamma' = \Gamma[r_z \mapsto\ \langle O, int, 0\rangle][r_{\mathsf{i}} \mapsto\ \langle B, ok, e_\ell\rangle] \qquad \Delta; \Psi; \Gamma'; \sigma \circ e_\ell; e_\ell; \tau\ opt \vdash b\end{array}}{(\Delta, x : \kappa_{int}); \Psi; \Gamma; \sigma \circ e_\ell; x_i; \tau\ opt \vdash \texttt{recovernz}\ r_z;\ b}\ (recovernz\text{-}t)$$

$$\frac{\begin{array}{lll}\Gamma(r_i) = \langle B, go, e_t'\rangle & \Gamma(r_t) = \langle G, \forall[\Delta_t](\Gamma_t, \sigma_t), e_t\rangle & \Delta \vdash e_t = e_t' \\ \exists S_t\ .\ \Delta \vdash S_t : \Delta_t & \Delta \vdash \Gamma[r_{\mathsf{i}} \mapsto \langle O, check, e_t'\rangle] \leq S_t(\Gamma_t) & \Delta \vdash \sigma \circ e_\ell \circ e_t = S_t(\sigma_t)\end{array}}{\Delta; \Psi; \Gamma; \sigma \circ e_\ell; e_i; t \vdash \texttt{jmp}\ r_t}\ (jmp\text{-}t)$$

Fig. 3. Selected Instruction Typing Rules and Block Typing Rules

to ℓ, then $\Delta \vdash e_i = \ell$. The option type $\tau\ opt$ contains the type of the label $\ell + 1$ if such a label exists. It is used when a branch falls through to the subsequent block to determine the type of that block. Three example rules are shown in Figure 3.

The first rule, *sequence-t*, is used when the first instruction in a block is one of the basic instructions described previously. The second rule for checking blocks illustrates how to check the `recovernz` instruction. At run time, control only proceeds past this point in the block if x_i (describing r_{i}) is equal to the expression e_ℓ (describing the current location), so the remainder of the block is typed by substituting e_ℓ for x_i. The types of r_{i} and r_z are updated to reflect the deletion of x_i. Judgment $\Delta \vdash \Gamma/r_{\mathsf{i}}/r_z$ wf and $\Delta \vdash \sigma$ wf hold when all expression variables used in the types of registers other than r_{i} and r_z, as well as in the expressions in σ, are all contained in Δ. Since none of these pieces of state contain x_i, they do not need to be modified.

The rule *jmp-t* requires that r_{i} has type $\langle B, go, e_t'\rangle$ specifying that the intention must already have been set before the jump. Also, the actual jump target in r_t has a code type and is described by an expression e_t that is equal to e_t'. This enforces that in the absence of faults, the duplicate target is equal to the target. The target label precondition contains a set of expression variables Δ_t

and requires a register file described by Γ_t and a history described by σ_t. There is some substitution S_t for the variables in Δ_t so that the current register file type and sequence are subtypes of those required by the target. The `jmp` r_t instruction recolors the blue intention register to be orange when control is transferred to a new block. At first, this seems to contradict the rule that faults to a value of one color should never corrupt values of other colors. However, because the target block doesn't place any restrictions on the expression describing r_{i}, the variable x_i that describes the value can be instantiated with the value itself. Because of this, a blue value that is not trusted can become a trusted orange value during a control flow transfer, continuing to leave only the blue values untrusted.

5 Formal Properties

We have proven a number of properties of our type system including variants of the standard Progress, Preservation and Type Safety theorems. Our most important result is a Fault Tolerance theorem, which we sketch briefly below. The full proofs appear in the online appendix [13].

In order to explain the theorem, we require a couple of additional concepts. First, we say a machine state Σ is well-formed (written $\vdash^Z \Sigma$) when all code and state are well-typed relative to the zap tag Z. Second, we say a faulty machine state Σ_f simulates a fault-free state Σ under color c (written $\Sigma_f \overset{c}{\sim} \Sigma$) whenever the two states are identical modulo values colored c. In other words, values colored c may be completely different from one another, but otherwise the two states are identical.

The judgment $\Sigma \Longrightarrow_k^h \mathcal{F}$ states that machine state Σ executes through a sequence of blocks h to reach state $\mathcal{F}$ while incurring k faulty transitions. So if $\Sigma = (C, h_1, R, b)$, then $\mathcal{F}$ is either $(C, (h_1, h), R', b')$, $\texttt{hwerror}(h_1, h)$, or $\texttt{recover}(h_1, h)$.

We say a program is fault-tolerant if any execution of the program with a single fault behaves in one of four possible ways with regards to the original, non-faulty computation: (1) The faulty computation visits the same sequence of blocks as the original, and the final faulty state simulates the original result state under some color c. (2) The faulty computation attempts to transfer control to an invalid address outside the domain of code memory and triggers a hardware fault. Prior to the occurrence of the hardware fault, the faulty computation visited the same blocks as the original computation. (3) A fault affecting the intentions register or checking code cause the faulty computation to conservatively detect a fault in software and jump to recovery code. (4) The faulty computation veers off course to a block that does not match the corresponding block in the original computation. In this case, the checking code in the invalid block catches the error and transfers control to the recovery code.

Theorem 1 (Fault Tolerance). *If* $\vdash \Sigma$ *and* $\Sigma \Longrightarrow_0^h \Sigma'$ *then at least one of the following cases applies and all derivations* $\Sigma \Longrightarrow_1^{h_f} \mathcal{F}$ *where* $\mathit{length}(h_f) \leq \mathit{length}(h)$ *fit one of these cases:*

1. $\Sigma \Longrightarrow_1^h \Sigma'_f$ *and* $\exists c \,.\, \Sigma'_f \overset{c}{\sim} \Sigma'$
2. $\Sigma \Longrightarrow_1^{h_f}$ $\texttt{hwerror}(h', h_f)$ *and* h_f *is a prefix of* h
3. $\Sigma \Longrightarrow_1^{h_f}$ $\texttt{recover}(h', h_f)$ *and* h_f *is a prefix of* h
4. $\Sigma \Longrightarrow_1^{h_f}$ $\texttt{recover}(h', h_f)$ *and* $h_f = (h_1, l')$ *and* $h = (h_1, l, h_2)$

6 Related Work, Future Work, and Conclusions

Related Work. As mentioned in the introduction, this research follows previous work on λ_{zap} [18] and TAL$_{\text{FT}}$ [11]. However, neither λ_{zap} nor TAL$_{\text{FT}}$ provided software mechanisms for guaranteeing control-flow integrity. Recently, Elsman [7] has shown how to extend λ_{zap} so that the atomic voting operations can be broken down into a series of conditional statements. However, again, there is no treatment of control-flow.

Perhaps the most closely related work to the current paper is CFI, a provably-sound technique for enforcing control-flow integrity in a security context [1,2]. The goal of CFI is to guarantee that machine code obeys a predefined "control-flow policy" that constrains the sequence of blocks control can move through. The key distinction between CFI and our own work is the threat model. CFI attackers can modify arbitrary amounts of machine state in arbitrary ways. but cannot touch three reserved registers during the execution of certain code sequences.

Our work builds upon many past research efforts in fault tolerance, particularly those that deal with control-flow checking. For example, Oh *et al.* [10] developed a pure software control-flow checking scheme (CFCSS) wherein each control transfer generates a run-time signature that is validated by error checking code generated by the compiler for every block. The SWIFT system [15], another software-only fault tolerance system, also uses signature checking very much like that in the current paper. The distinguishing feature of our research is not the control-flow checking procedure itself, but the type system we designed to verify the code and our proof that well-typed programs are indeed fault tolerant. These previous efforts did not rigorously specify the properties they intended to enforce nor did they prove their techniques actually enforce them.

Future Work. We acknowledge that the fault model used in this paper is simplistic. By assuming hardware support to catch control transfers into the middle of blocks, we avoid dealing with many interesting and likely situations. This assumption is required because stating intentions involves resetting r_{i}, so an incorrect transfer into a block before the `intend` r_t instruction may not be caught.

A sequence of existing work on software-only solutions [10,15,5] handles increasing classes of erroneous transfers. By ensuring that the intentions register is a function of the entire control-flow path, not just the current block, they can detect most jumps into the middle of blocks. For example, SWIFT [15] keeps an "approximate program counter" which contains the current block. Before each control-flow transfer, the current block and the intended target block are xored together and put in a designated "transition register". At the beginning of each block, the transition register is xored with the approximate program counter to give the new approximation. (A correct transfer from block A to block B will result in a new approximate program counter of $A \otimes (A \otimes B)$, which is equal to B.) Though these solutions are an improvement, there are still situations (such as jumping back two instructions within a block) that they cannot handle.

The classification scheme of values and reliability properties from this paper does not transfer directly to these more complex solutions, but we believe we can develop a similar classification to capture the necessary invariants. In doing so, the Fault Tolerance Theorem becomes more difficult to state and prove due to the increase of possible scenarios a single fault may cause. (For example, a fault may cause control to transfer from the middle of one block to the middle of a second block. This second block may transfer control to a third block before the error is finally detected.) In essence, the current work and proof strategy are an important building block for reasoning about more complex solutions.

Conclusions. Future processors will become more susceptible to transient faults, and reasoning about the correctness of software running on faulty hardware is an extremely difficult task, particularly when faults may affect program control flow. In this paper, we defined a simple abstract machine that exhibits control-flow faults and we analyzed the correctness of a software protocol for detecting them. Our analysis proceeded through the definition of a type system that guarantees programs are reliable relative to a simple fault model. We have rigorously proven strong reliability properties for our type system and believe this is the first successful attempt at reasoning rigorously about software mechanisms for controlling control flow faults.

References

1. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity: Principles, implementations, and applications. In: ACM Conference on Computer and Communications Security (November 2005)
2. Abadi, M., Budiu, M.: A theory of secure control flow. In: International Conference on Formal Engineering Methods (November 2005)
3. Baumann, R.C.: Soft errors in advanced semiconductor devices-part I: the three radiation sources. IEEE Transactions on Device and Materials Reliability 1(1), 17–22 (2001)
4. Baumann, R.C.: Soft errors in commercial semiconductor technology: Overview and scaling trends. In: IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals, pp. 121_01.1–121_01.14 (April 2002)
5. Borin, E., Wang, C., Wu, Y., Araujo, G.: Software-based transparent and comprehensive control-flow error detection. In: CGO 2006: Proceedings of the International Symposium on Code Generation and Optimization, Washington, DC, USA, pp. 333–345. IEEE Computer Society Press, Los Alamitos (2006)
6. Borkar, S.: Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. In: IEEE Micro., vol. 25, pp. 10–16 (December 2005)

7. Elsman, M.: Fault-tolerant voting in a simply-typed lambda calculus. Technical Report ITU-TR-2007-99, IT University of Copenhagen, Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark (June 2007)
8. Michalak, S.E., Harris, K.W., Hengartner, N.W., Takala, B.E., Wender, S.A.: Predicting the number of fatal soft errors in Los Alamos National Labratory's ASC Q computer. IEEE Transactions on Device and Materials Reliability 5(3), 329–335 (2005)
9. O'Gorman, T.J., Ross, J.M., Taber, A.H., Ziegler, J.F., Muhlfeld, H.P., Montrose, I.C.J., Curtis, H.W., Walsh, J.L.: Field testing for cosmic ray soft errors in semiconductor memories. IBM Journal of Research and Development, 41–49 (January 1996)
10. Oh, N., Shirvani, P.P., McCluskey, E.J.: Control-flow checking by software signatures. In: IEEE Transactions on Reliability, vol. 51, pp. 111–122 (March 2002)
11. Perry, F., Mackey, L., Reis, G.A., Ligatti, J., August, D.I., Walker, D.: Fault-tolerant typed assembly language. In: International Symposium on Programming Language Design and Implementation (PLDI) (June 2007)
12. Perry, F., Walker, D.: Reasoning about control flow in the presence of transient faults. Technical Report TR-799-07, Princeton University (2007)
13. Perry, F., Walker, D.: Reasoning about control flow in the presence of transient faults - online proof appendix (2007), Web site: http://www.cs.princeton.edu/sip/projects/zap/tal_cf/
14. Reinhardt, S.K., Mukherjee, S.S.: Transient fault detection via simultaneous multithreading. In: Proceedings of the 27th Annual International Symposium on Computer Architecture, pp. 25–36. ACM Press, New York (2000)
15. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I.: SWIFT: Software implemented fault tolerance. In: Proceedings of the 3rd International Symposium on Code Generation and Optimization (March 2005)
16. Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., August, D.I., Mukherjee, S.S.: Design and evaluation of hybrid fault-detection systems. In: Proceedings of the 32th Annual International Symposium on Computer Architecture, pp. 148–159 (June 2005)
17. Shivakumar, P., Kistler, M., Keckler, S.W., Burger, D., Alvisi, L.: Modeling the effect of technology trends on the soft error rate of combinational logic. In: Proceedings of the 2002 International Conference on Dependable Systems and Networks, pp. 389–399 (June 2002)
18. Walker, D., Mackey, L., Ligatti, J., Reis, G., August, D.I.: Static typing for a faulty lambda calculus. In: ACM International Conference on Functional Programming, Portland, Oregon (September 2006)

A Calculational Approach to Control-Flow Analysis by Abstract Interpretation

Jan Midtgaard[1] and Thomas Jensen[2]

[1] INRIA Rennes - Bretagne Atlantique
[2] CNRS
IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
{jan.midtgaard,thomas.jensen}@irisa.fr

Abstract. We present a derivation of a control-flow analysis by abstract interpretation. Our starting point is a transition system semantics defined as an abstract machine for a small functional language in continuation-passing style. We obtain a Galois connection for abstracting the machine states by composing Galois connections, most notable an independent-attribute Galois connection on machine states and a Galois connection induced by a closure operator associated with a constituent-parts relation on environments. We calculate abstract transfer functions by applying the state abstraction to the collecting semantics, resulting in a novel characterization of demand-driven 0-CFA.

1 Introduction

Over twenty-five years ago Jones [16] statically approximated the flow of lambda-expressions. Since then control-flow analysis (CFA) has been the subject of immense research [2,31,30,25]. Ten years ago Nielson and Nielson designed a co-inductive collecting semantics for control-flow analysis and at the same time asked [25, p.1]: *How does one exploit Galois connections and widenings to systematically coarsen [control-flow analysis]?"*

In this paper we take the first steps towards answering that question, by expressing a control-flow analysis as the composition of several well-known Galois connections, thereby spelling out the approximations taking place. Our approach thus follows Cousot's programme of *calculational abstract interpretation* [6] in which an abstract interpretation is calculated by systematically applying abstraction functions to a formal programming language semantics.

We develop our approach in the setting of CFA for functional languages. A substantial amount of work concerned with abstract interpretation of functional languages is based on denotational semantics [19] in which source-level functions are modelled with mathematical functions [29]. However, as CFA is concerned with operational information about source-level functions we believe that a denotational starting point is inadequate for a calculational derivation of control-flow analysis by abstract interpretation. Instead, we have chosen to base our derivation on an operational semantics in the form of an abstract machine (a transition system) in which source-level functions are modelled with *closures* which are pairs of expressions and environments. Closures were originally suggested by Landin to model functions in the SECD machine [20] and have since become a standard implementation method for functional languages [1].

© Springer-Verlag Berlin Heidelberg 2008

$$\begin{tikzcd}
\wp(\mathit{SExp} \times \mathit{Env}) \arrow[d, shift right, "\alpha_\times"'] & \text{collecting semantics} & \text{(Figure 4)} \\
\wp(\mathit{SExp}) \times \wp(\mathit{Env}) \arrow[u, shift right, "\gamma_\times"'] \arrow[d, shift right, "\alpha_\otimes"'] & \text{intermediate transfer function} & \text{(Figure 5)} \\
\wp(\mathit{SExp}) \times \mathit{Env}^\sharp \arrow[u, shift right, "\gamma_\otimes"'] & \text{demand-driven 0-CFA} & \text{(Figure 6)}
\end{tikzcd}$$

Fig. 1. The big picture

The aim of a functional CFA is to determine which functions are bound to which variables during execution. This information is found in the environments during the execution of the program. Accordingly, an essential part of the abstraction technique that we propose is to express appropriate Galois connections for extracting the approximate environment components of a machine state. Environments are recursive structures that themselves may contain closures which in turn contain environments. A crucial step in the derivation is the definition of an upper closure operator on environments that induce an appropriate Galois connection. Figure 1 summarizes the two steps of the abstraction. From a reachable-states collecting semantics of the CE machine [14], defined in Section 5, we first abstract the machine components as independent attributes. Next we abstract the pair of sets by an environment abstraction based on the notion of *constituent relation* of Milner and Tofte [23]. These and other Galois connections for abstracting values are defined in Section 6. The composition of these Galois connections yields an abstraction function that is again applied to the transfer function to calculate a demand-driven 0-CFA. The calculations are given in Section 7. The contributions of the paper are as follows.

– We start from a well-known operational semantics, instead of instrumenting a collecting semantics.
– We apply Cousot-style abstract interpretation to CFA: reachable states of a transition system systematically abstracted through Galois connections.
– We explain the approximations of a CFA by spelling it out as the composition of several well-known Galois connections.
– We characterize demand-driven 0-CFA as a simple independent attribute abstraction of a reachable states semantics.
– Finally, we *calculate* the analysis rather than postulating it and verifying it a posteriori.

An implementation of the derived analysis and an example is briefly explained in Section 8. Section 9 compares the analysis to related work. Section 10 concludes and lists future work which notably consists in calculating a flow-sensitive CFA by using alternative Galois connections to approximate machine states. We assume the reader is familiar with operational semantics, continuation-passing style (CPS), control-flow analysis, complete lattices, and fixed points. The following sections provides a concise summary of well-known facts about abstract interpretation that will be used in the paper.

2 A Short Introduction to Abstract Interpretation

We recall a number of properties related to complete lattices, Galois connections, and closure operators. The section consists of known material from the research literature [8,9,11,7,13]. Readers familiar with the material can skip to the next section.

2.1 Galois Connections etc.

The powerset of a set S is written $\wp(S)$. The powerset $\wp(S)$ ordered by set inclusion is a complete lattice $\langle \wp(S); \subseteq, \emptyset, S, \cup, \cap \rangle$. A *Galois connection* between two posets $\langle \mathcal{D}_1; \subseteq_1 \rangle$ and $\langle \mathcal{D}_2; \subseteq_2 \rangle$ (the *concrete* and the *abstract* domain) is a pair of maps $\alpha : \mathcal{D}_1 \to \mathcal{D}_2$ (the *abstraction* map) and $\gamma : \mathcal{D}_2 \to \mathcal{D}_1$ (the *concretisation* map) such that $\forall c \in \mathcal{D}_1 : \forall a \in \mathcal{D}_2 : \alpha(c) \subseteq_2 a \iff c \subseteq_1 \gamma(a)$. Equivalently α is monotone, γ is monotone, $\gamma \circ \alpha$ is extensive ($\forall c \in \mathcal{D}_1 : c \subseteq_1 \gamma \circ \alpha(c)$), and $\alpha \circ \gamma$ is reductive ($\forall a \in \mathcal{D}_2 : \alpha \circ \gamma(a) \subseteq_2 a$). Galois connections are typeset as $\langle \mathcal{D}_1; \subseteq_1 \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathcal{D}_2; \subseteq_2 \rangle$. When the domains are apparent from the context we use the lighter notation $\mathcal{D}_1 \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}_2$. Now let $\langle \mathcal{D}_1; \subseteq_1, \bot_1, \top_1, \cup_1, \cap_1 \rangle$ and $\langle \mathcal{D}_2; \subseteq_2, \bot_2, \top_2, \cup_2, \cap_2 \rangle$ be complete lattices. Given a Galois connection $\mathcal{D}_1 \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}_2$ then α is a complete join-morphism (CJM) ($\alpha(\cup_1 X) = \cup_2 \alpha(X) = \cup_2\{\alpha(x) \mid x \in X\}$) (and γ is a complete meet-morphism). Given a complete join-morphism α and $\gamma(y) = \cup_1\{x \in \mathcal{D}_1 \mid \alpha(x) \subseteq_2 y\}$ then they form a Galois connection.

Example 1 (Identity abstraction). Two identity functions $1_{\mathcal{D}} = \lambda x.\, x$ form a Galois connection on a poset $\langle \mathcal{D}; \subseteq \rangle \underset{1_{\mathcal{D}}}{\overset{1_{\mathcal{D}}}{\leftrightarrows}} \langle \mathcal{D}; \subseteq \rangle$.

Example 2 (Elementwise abstraction). Let an elementwise operator $@ : C \to A$ be given. Define $\alpha_@(P) = \{@(p) \mid p \in P\}$ and $\gamma_@(Q) = \{p \mid @(p) \in Q\}$. Then $\wp(C) \underset{\alpha_@}{\overset{\gamma_@}{\leftrightarrows}} \wp(A)$.

Example 3 (Pointwise abstraction of a set of functions). Assume an abstraction

$$\langle \wp(\mathcal{D}_2); \subseteq, \emptyset, \mathcal{D}_2, \cup, \cap \rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle \mathcal{D}_2^\sharp; \subseteq_2^\sharp, \bot_2^\sharp, \top_2^\sharp, \cup_2^\sharp, \cap_2^\sharp \rangle$$

and let

$$\begin{aligned} \alpha_\Pi(F) &= \lambda x.\, \alpha_2(\{f(x) \mid x \in \mathcal{D}_1 \ \wedge \ f \in F\}) \\ \gamma_\Pi(\Phi) &= \{f \in \mathcal{D}_1 \to \mathcal{D}_2 \mid \forall x : f(x) \in \gamma_2(\Phi(x))\} \end{aligned}$$

Then $\langle \wp(\mathcal{D}_1 \to \mathcal{D}_2); \subseteq, \emptyset, \mathcal{D}_1 \to \mathcal{D}_2, \cup, \cap \rangle \underset{\alpha_\Pi}{\overset{\gamma_\Pi}{\leftrightarrows}} \langle \mathcal{D}_1 \to \mathcal{D}_2^\sharp; \dot{\subseteq}_2^\sharp, \dot{\bot}_2^\sharp, \dot{\top}_2^\sharp, \dot{\cup}_2^\sharp, \dot{\cap}_2^\sharp \rangle$ where we have used the pointwise notation $A \ \dot{r}\ B \iff \forall x.A(x)\ r\ B(x)$ and $\dot{c} = \lambda\_.\, c$ for relations and constants.

Cousot and Cousot [11, Sect.3] describe the pointwise abstraction as the composition of three abstractions.

One can abstract a set of pairs into a pair of sets, in turn performing an *attribute independent abstraction* [18], as relational information between the components of the individual pairs is lost.

Example 4 (Abstraction of a binary relation by a pair of sets). Let

$$\alpha_\times(r) = \langle \pi_1(r),\, \pi_2(r)\rangle \qquad\qquad \gamma_\times(\langle X,\, Y\rangle) = X \times Y$$

where $\pi_1(r) = \{x \mid \exists y : \langle x,\, y\rangle \in r\}$, $\pi_2(r) = \{y \mid \exists x : \langle x,\, y\rangle \in r\}$, and let $\subseteq_\times = \subseteq \times \subseteq$, $\bot_\times = \langle \emptyset,\, \emptyset\rangle$, $\top_\times = \langle \mathcal{D}_1,\, \mathcal{D}_2\rangle$, $\cup_\times = \cup \times \cup$, and $\cap_\times = \cap \times \cap$. Then

$$\langle \wp(\mathcal{D}_1 \times \mathcal{D}_2); \subseteq, \emptyset, \mathcal{D}_1 \times \mathcal{D}_2, \cup, \cap\rangle \underset{\alpha_\times}{\overset{\gamma_\times}{\leftrightarrows}} \langle \wp(\mathcal{D}_1) \times \wp(\mathcal{D}_2); \subseteq_\times, \bot_\times, \top_\times, \cup_\times, \cap_\times\rangle$$

An *upper closure operator* is a map $\rho : \mathcal{D} \to \mathcal{D}$ on a poset $\langle \mathcal{D}; \subseteq\rangle$ that is extensive ($\forall x \in \mathcal{D} : x \subseteq \rho(x)$), monotone ($\forall x, x' \in \mathcal{D} : x \subseteq x' \implies \rho(x) \subseteq \rho(x')$), and idempotent ($\forall x \in \mathcal{D} : \rho(x) = \rho(\rho(x))$). A closure operator ρ on a poset $\langle \mathcal{D}; \subseteq\rangle$ induces a Galois connection: $\langle \mathcal{D}; \subseteq\rangle \underset{\rho}{\overset{1_{\mathcal{D}}}{\leftrightarrows}} \langle \rho(\mathcal{D}); \subseteq\rangle$. Finally the image of a complete lattice $\langle \mathcal{D}; \subseteq, \bot, \top, \cup, \cap\rangle$ by a closure operator ρ is itself a complete lattice $\langle \rho(\mathcal{D}); \subseteq, \rho(\bot), \rho(\top), \lambda X.\, \rho(\cup X), \cap\rangle$.

2.2 Composition of Galois Connections

Galois connections enjoy a number of properties regarding composition. One can abstract a pair of sets by abstracting its components.

Example 5 (Abstraction of a pair of sets by an abstract pair). Assuming two Galois connections $\langle \wp(\mathcal{D}_1); \subseteq, \emptyset, \mathcal{D}_1, \cup, \cap\rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle \mathcal{D}_1^\sharp; \subseteq_1^\sharp, \bot_1^\sharp, \top_1^\sharp, \cup_1^\sharp, \cap_1^\sharp\rangle$ and $\langle \wp(\mathcal{D}_2); \subseteq, \emptyset, \mathcal{D}_2, \cup, \cap\rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle \mathcal{D}_2^\sharp; \subseteq_2^\sharp, \bot_2^\sharp, \top_2^\sharp, \cup_2^\sharp, \cap_2^\sharp\rangle$, define

$$\alpha_\otimes(\langle X,\, Y\rangle) = \langle \alpha_1(X),\, \alpha_2(Y)\rangle \qquad\qquad \gamma_\otimes(\langle x,\, y\rangle) = \langle \gamma_1(x),\, \gamma_2(y)\rangle$$

where $\subseteq_\otimes = \subseteq_1^\sharp \times \subseteq_2^\sharp$, $\bot_\otimes = \langle \bot_1^\sharp,\, \bot_2^\sharp\rangle$, $\top_\otimes = \langle \top_1^\sharp,\, \top_2^\sharp\rangle$, $\cup_\otimes = \cup_1^\sharp \times \cup_2^\sharp$, and $\cap_\otimes = \cap_1^\sharp \times \cap_2^\sharp$. Then

$$\langle \wp(\mathcal{D}_1) \times \wp(\mathcal{D}_2); \subseteq_\times, \bot_\times, \top_\times, \cup_\times, \cap_\times\rangle \underset{\alpha_\otimes}{\overset{\gamma_\otimes}{\leftrightarrows}} \langle \mathcal{D}_1^\sharp \times \mathcal{D}_2^\sharp; \subseteq_\otimes, \bot_\otimes, \top_\otimes, \cup_\otimes, \cap_\otimes\rangle$$

Most importantly Galois connections compose sequentially.

Lemma 1 (Compositional abstraction). *Given two Galois connections between complete lattices* $\langle \mathcal{D}_0; \subseteq_0, \bot_0, \top_0, \cup_0, \cap_0\rangle \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \langle \mathcal{D}_1; \subseteq_1, \bot_1, \top_1, \cup_1, \cap_1\rangle$ *and* $\langle \mathcal{D}_1; \subseteq_1, \bot_1, \top_1, \cup_1, \cap_1\rangle \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \langle \mathcal{D}_2; \subseteq_2, \bot_2, \top_2, \cup_2, \cap_2\rangle$, *then*

$$\langle \mathcal{D}_0; \subseteq_0, \bot_0, \top_0, \cup_0, \cap_0\rangle \underset{\alpha_2 \circ \alpha_1}{\overset{\gamma_1 \circ \gamma_2}{\leftrightarrows}} \langle \mathcal{D}_2; \subseteq_2, \bot_2, \top_2, \cup_2, \cap_2\rangle$$

3 Language

As our source language we take CPS expressions characterized by the grammar in Figure 2 [12]. The grammar distinguishes between *serious* expressions, denoting expressions whose evaluation may diverge, and *trivial* expressions, denoting

$$\begin{array}{lll}
p ::= \lambda k.\, e & & \text{(CPS programs)} \\
e ::= t_0\, t_1\, c \mid c\, t & & \text{(serious CPS expressions)} \\
t ::= x \mid v \mid \lambda x, k.\, e & & \text{(trivial CPS expressions)} \\
c ::= \lambda v.\, e \mid k & & \text{(continuation expressions)}
\end{array}$$

Fig. 2. BNF of CPS language

expressions whose evaluation will terminate. We further distinguish three different forms of variables: source variables x, continuation variables k, and formal continuation parameters v, each drawn from disjoint countable sets of variables X, K, and V, respectively. We let *Var* denote the disjoint union of the three $\mathit{Var} = X \cup V \cup K$. When apparent from the context we will also use x as a generic meta-variable $x \in \mathit{Var}$.

For brevity we write $\lambda x, k.\, e$ for $\lambda x.\, \lambda k.\, e$. Furthermore we let *SExp*, *TExp*, and *CExp* denote the domain of serious expressions $\mathit{SExp} = \mathcal{L}(e)$, the domain of trivial expressions $\mathit{TExp} = \mathcal{L}(t)$, and the domain of continuation expressions $\mathit{CExp} = \mathcal{L}(c)$, respectively.[1] Slightly misusing notation we will furthermore use e, t, and c (with subscripts and primes) as meta-variables to denote a serious expression, a trivial expression, and a continuation expression, respectively. Their use will be apparent from the context. The language is Turing-complete, in that it is sufficient to express a CPS-version of the Ω-combinator $(\lambda x.\, x\, x\, \lambda y.\, y\, y)$: $\lambda k_0.\, (\lambda x, k_1.\, x\, x\, k_1)\, (\lambda y, k_2.\, y\, y\, k_2)\, k_0$.

4 Semantics

Our starting point semantics will be the CE machine of Flanagan et al. [14]. As opposed to Flanagan et al. we consider only unary functions in the CPS language. Furthermore, our starting-point grammar is a slightly refactored version of the grammar given by Flanagan et al., and the machine description has been refactored accordingly. As a consequence the number of machine transitions is thus cut down to two.

The values and environments of the machine are given in Figure 3a. Values can be either closures, continuation closures, or a special `stop` value that signals a machine halt. We let *Val* denote the set of values $\mathit{Val} = \mathcal{L}(w)$ and we let *Env* denote the set of environments $\mathit{Env} = \mathcal{L}(r)$. The environments in *Env* constitute partial functions, with $\bullet$ being the partial function nowhere defined. We use w and r (with subscripts and primes) as meta-variables to denote a machine value and a machine environment, respectively. Again their use will be apparent from the context. In the spirit of the original CE machines helper function μ, we formulate two helper functions μ_t and μ_c in Figure 3b for evaluating trivial expressions and continuation expressions, respectively.

A machine state is a pair consisting of a serious expression and an environment. The transition relation of the machine is given in Figure 3c. The initial

[1] Using the $\mathcal{L}(N)$ notation for the language generated by the non-terminal N.

$$\begin{array}{llr} w ::= [\lambda x, k.\, e,\ r] \ \mid\ [\lambda v.\, e,\ r] \ \mid\ \texttt{stop} & & \text{(values)} \\ r\ ::= \bullet \ \mid\ r[x \mapsto w] & & \text{(environments)} \end{array}$$

(a) Values and environments

$$\begin{aligned} \mu_t &: TExp \times Env \rightharpoonup Val \\ \mu_t(x, r) &= r(x) \\ \mu_t(v, r) &= r(v) \\ \mu_t(\lambda x, k.\, e, r) &= [\lambda x, k.\, e,\ r] \end{aligned} \qquad \begin{aligned} \mu_c &: CExp \times Env \rightharpoonup Val \\ \mu_c(k, r) &= r(k) \\ \mu_c(\lambda v.\, e, r) &= [\lambda v.\, e,\ r] \end{aligned}$$

(b) Helper functions

$$\langle t_0\, t_1\, c,\ r\rangle \longrightarrow \langle e,\ r'[x \mapsto w][k \mapsto w_c]\rangle \qquad \text{where } \begin{aligned} [\lambda x, k.\, e,\ r'] &= \mu_t(t_0, r) \\ w &= \mu_t(t_1, r) \\ w_c &= \mu_c(c, r) \end{aligned}$$

$$\langle c\, t,\ r\rangle \longrightarrow \langle e,\ r'[v \mapsto w]\rangle \qquad \text{where } \begin{aligned} [\lambda v.\, e,\ r'] &= \mu_c(c, r) \\ w &= \mu_t(t, r) \end{aligned}$$

(c) Transition relation

$$\begin{aligned} & eval(\lambda k.\, e) = w \ \text{ iff} \\ & \langle e,\ \bullet[k \mapsto [\lambda v_r.\, k_r\, v_r,\ \bullet[k_r \mapsto \texttt{stop}]]]\rangle \longrightarrow^* \langle k_r\, v_r,\ \bullet[k_r \mapsto \texttt{stop}][v_r \mapsto w]\rangle \end{aligned}$$

(d) Machine evaluation

Fig. 3. The CE abstract machine

state of the machine binds the initial continuation variable k to a special continuation closure containing a `stop` value. When applied, the special continuation closure will first bind the final result to a special variable v_r, and afterwards attempt to apply the `stop` value (the latter indicating a final state). The machine evaluates CPS programs by repeatedly transitioning from state to state until it is either stuck or in a final state.

5 Collecting Semantics

As traditional we consider the reachable states of the transition system as our collecting semantics [5,10].

$$\begin{aligned} I_{\lambda k.\, e} &= \{\langle e,\ \bullet[k \mapsto [\lambda v_r.\, k_r\, v_r,\ \bullet[k_r \mapsto \texttt{stop}]]]\rangle\} && \text{(initial state)} \\ F_{\lambda k.\, e} &: \wp(SExp \times Env) \to \wp(SExp \times Env) \\ F_{\lambda k.\, e}(S) &= I_{\lambda k.\, e} \cup \{s \mid \exists s' \in S : s' \longrightarrow s\} && \text{(strongest post-condition)} \end{aligned}$$

The reachable states semantics is now given as the limit $\bigcup_{n \geq 0} F^n_{\lambda k.\, e}(\emptyset)$. Due to the notational overhead we shall refrain from subscripting with the program at hand from here onwards.

$$\begin{aligned}
\mu_t^{\wp} &: \mathit{TExp} \times \wp(\mathit{Env}) \to \wp(\mathit{Val}) \\
\mu_t^{\wp}(x, R) &= \{r(x) \mid r \in R\} \\
\mu_t^{\wp}(v, R) &= \{r(v) \mid r \in R\} \\
\mu_t^{\wp}(\lambda x, k.\, e, R) &= \{[\lambda x, k.\, e,\, r] \mid r \in R\}
\end{aligned}
\qquad
\begin{aligned}
\mu_c^{\wp} &: \mathit{CExp} \times \wp(\mathit{Env}) \to \wp(\mathit{Val}) \\
\mu_c^{\wp}(k, R) &= \{r(k) \mid r \in R\} \\
\mu_c^{\wp}(\lambda v.\, e, R) &= \{[\lambda v.\, e,\, r] \mid r \in R\}
\end{aligned}$$

(a) Collecting helper functions

$$\begin{aligned}
F_c &: \wp(\mathit{SExp} \times \mathit{Env}) \to \wp(\mathit{SExp} \times \mathit{Env}) \\
F_c(S) &= I_{\lambda k.\, e} \\
&\cup \{\langle e',\, r'[x \mapsto w][k' \mapsto w_c]\rangle \mid \exists \langle t_0\, t_1\, c,\, r\rangle \in S : \\
&\qquad [\lambda x, k'.\, e',\, r'] \in \mu_t^{\wp}(t_0, \{r\}) \\
&\qquad \wedge\ w \in \mu_t^{\wp}(t_1, \{r\}) \\
&\qquad \wedge\ w_c \in \mu_c^{\wp}(c, \{r\})\} \\
&\cup \{\langle e',\, r'[v \mapsto w]\rangle \mid \exists \langle c\, t,\, r\rangle \in S : \\
&\qquad [\lambda v.\, e',\, r'] \in \mu_c^{\wp}(c, \{r\}) \\
&\qquad \wedge\ w \in \mu_t^{\wp}(t, \{r\})\}
\end{aligned}$$

(b) Transition function

Fig. 4. Reachable states collecting semantics

We give an equivalent formulation of the collecting semantics in Figure 4 with helper functions extended to operate on sets of environments. A simple case analysis reveals that $\mu_t^{\wp}$ and $\mu_c^{\wp}$ are monotone in their second argument. By another case analysis one can establish two equivalences between the helper functions $\forall t, R : \forall r \in R : \{w \mid w = \mu_t(t, r)\} = \mu_t^{\wp}(t, \{r\})$ and $\forall c, R : \forall r \in R : \{w \mid w = \mu_c(c, r)\} = \mu_c^{\wp}(c, \{r\})$. A final case analysis establishes the equivalence of the two: $\forall S : \langle e,\, r\rangle \in F(S) \iff \langle e,\, r\rangle \in F_c(S)$.

6 Abstracting the Collecting Semantics

With the collecting semantics in place we are now in position to abstract it. We describe the abstractions for values, environments, and machine states in turn.

6.1 Abstraction of Values

Values are abstracted using the elementwise abstraction of Example 2. First, the grammar of abstract values reads as follows.

$$w^{\sharp} ::= [\lambda x, k.\, e] \mid [\lambda v.\, e] \mid \mathtt{stop} \quad \text{(abstract values)}$$

We let $\mathit{Val}^{\sharp} = \mathcal{L}(w^{\sharp})$ denote the domain of abstract values. Secondly, we define an elementwise operator mapping a concrete value to its abstract counterpart.

The operator abstracts away the captured environment component of closure values.

$$\begin{aligned} @ &: \mathit{Val} \rightarrow \mathit{Val}^\sharp \\ @([\lambda x, k.\, e,\ r]) &= [\lambda x, k.\, e] \\ @([\lambda v.\, e,\ r]) &= [\lambda v.\, e] \\ @(\mathtt{stop}) &= \mathtt{stop} \end{aligned}$$

6.2 Abstraction of Environments

In order to perform environment extension (binding) we need to concretize an environment component from an abstract closure. Unfortunately a straight-forward pointwise extension of the above value abstraction will not suffice: concretization of an abstract closure would return *top* representing *any* environment component. We therefore prefix the pointwise environment abstraction by an abstraction based on a closure operator to ensure that any captured environment in a set of environments belongs to the set itself. We will use a *constituent relation* formulated by Milner and Tofte [23] to express the closure operator.

For a tuple $(x_1, \ldots, x_n)$ each entry x_i is a *constituent* of the tuple. For a partial function $[x_1 \mapsto w_1 \ldots x_n \mapsto w_n]$, each w_i is a *constituent* of the function.[2] We write $x \succ y$ if y is a constituent of x. We denote by $\succ^*$ the reflexive transitive closure of the constituent relation.[3] We can now formulate an appropriate closure operator which induces the first Galois connection.

Definition 1 (Closure operator).

$$\begin{aligned} \rho &: \wp(\mathit{Env}) \rightarrow \wp(\mathit{Env}) \\ \rho(R) &= \{r' \in \mathit{Env} \mid \exists r \in R : r \succ^* r'\} \end{aligned}$$

Intuitively, given a set of environments, the closure operator returns a larger set containing all the "enclosed" environments of its argument. For the set of environments in the reachable states semantics the closure operator acts as an identity function, since the set is already closed.

Lemma 2 (ρ is an upper closure operator).
ρ is extensive, monotone, and idempotent.

Proof. The proofs for extensiveness and monotonicity are straightforward. Idempotency follows from extensiveness and transitivity of the $\succ^*$ relation.

Example 6 (Applying ρ). We apply ρ to the singleton environment $\{\bullet[k \mapsto [\lambda v_r.\, k_r\, v_r,\ \bullet[k_r \mapsto \mathtt{stop}]]]\}$ originating from the initial state $I_{\lambda k.\, e}$. Besides the element itself, $\bullet[k_r \mapsto \mathtt{stop}]$ is also a constituent environment. Hence

$$\begin{aligned} \rho(\{\bullet[k \mapsto [\lambda v_r.\, k_r\, v_r,\ \bullet[k_r \mapsto \mathtt{stop}]]]\}) = \{&\bullet[k_r \mapsto \mathtt{stop}], \\ &\bullet[k \mapsto [\lambda v_r.\, k_r\, v_r,\ \bullet[k_r \mapsto \mathtt{stop}]]]\} \end{aligned}$$

[2] Milner and Tofte [23] define the constituent relation on finite maps, whereas we define it for partial functions.

[3] Milner and Tofte [23] instead introduce constituent sequences.

Next we apply the pointwise abstraction from Example 3, based on the value abstraction of Section 6.1.

$$\rho(\wp(Env)) \underset{\alpha_\Pi}{\overset{\gamma_\Pi}{\leftrightarrows}} Env^\sharp \quad \text{where} \quad Env^\sharp = Var \to \wp(Val^\sharp)$$

Strictly speaking this Galois connection applies to the complete lattice $\wp(Env)$, whereas in this case we have a specialized complete lattice $\rho(\wp(Env))$. As the latter is a subset of the former, the definition of α_Π still applies. As for the existing definition of γ_Π its image $\wp(Env)$ does not agree with $\rho(\wp(Env))$. However since α_Π is a CJM it uniquely determines a specialized γ_Π. We leave a direct definition of γ_Π unspecified. With this in mind, we compose the two Galois connections and get another Galois connection: $\wp(Env) \underset{\rho}{\overset{1_{\wp(Env)}}{\leftrightarrows}} \rho(\wp(Env)) \underset{\alpha_\Pi}{\overset{\gamma_\Pi}{\leftrightarrows}} Env^\sharp$.

6.3 Abstraction of Machine States

As outlined in Figure 1 the abstraction of machine states is staged in two. We first abstract the independent attributes of the reachable states using Example 4. Next we abstract the pair of sets using Example 5 on the environment abstraction from Section 6.2. The first component, i.e., the set of reachable expressions, is not abstracted, hence instantiated with the identity abstraction.

As traditional [9,10,11] we consider the *reduced product* of both the abstract domains, i.e., all abstract pairs with an empty expression set or an empty abstract environment implicitly represent *bottom*.

7 Calculating the Analysis

We calculate an abstract transition function using a traditional recipe [10], by applying the independent attributes abstraction to the transition function from the collecting semantics. The following lemma determines the result as the best abstraction. It furthermore states the strategy for the calculation of a new transition function when read directionally from left to right. The resulting $F_\times$ appears in Figure 5.

Lemma 3 (Transition function as best abstraction)

$$\forall S : \alpha_\times(F_c(\gamma_\times(S))) = F_\times(S)$$

Proof Let $S = \langle E,\, R\rangle$ be given. Since $\alpha_\times$ is a complete join morphism, it distributes onto the three sets in F_c's definition. We consider the case of the last of the three sets:

$$\begin{aligned}
&\alpha_\times(\{\langle e',\, r'[v \mapsto w]\rangle \mid \exists\langle c\, t,\, r\rangle \in \gamma_\times(\langle E,\, R\rangle) : \\
&\qquad\qquad\qquad\qquad [\lambda v.\, e',\, r'] \in \mu_c^\wp(c, \{r\}) \\
&\qquad\qquad\qquad\qquad \wedge\ w \in \mu_t^\wp(t, \{r\})\}) \\
&= \alpha_\times\Big(\bigcup_{\substack{\langle c\, t, r\rangle \in \gamma_\times(\langle E, R\rangle) \\ [\lambda v.\, e', r'] \in \mu_c^\wp(c, \{r\}) \\ w \in \mu_t^\wp(t, \{r\})}} \{\langle e',\, r'[v \mapsto w]\rangle\}\Big) && \text{(formulate as join)}
\end{aligned}$$

$$= \bigcup_{\substack{\langle c\ t,\, r\rangle \in \gamma_\times(\langle E, R\rangle)\\ [\lambda v.\, e',\, r'] \in \mu_c^\wp(c,\{r\})\\ w \in \mu_t^\wp(t,\{r\})}}{}_\times\ \alpha_\times(\{\langle e',\ r'[v \mapsto w]\rangle\}) \qquad (\alpha_\times \text{ a CJM})$$

using the definitions of $\alpha_\times$ and $\gamma_\times$ we arrive at the last set of $F_\times$'s definition. □

Next we abstract the pair of sets by an abstract pair again following a recipe. We first calculate abstract helper functions $\mu_t^\sharp$ and $\mu_c^\sharp$. By construction they satisfy the following lemma. Furthermore the lemma states the strategy for the calculations when read directionally from left to right. The calculations proceed by case analysis.

Lemma 4 (Equivalence of helper functions)

$$\forall t, R : \alpha_@(\mu_t^\wp(t, R)) = \mu_t^\sharp(t, \alpha_\Pi(R)) \ \wedge\ \forall c, R : \alpha_@(\mu_c^\wp(c, R)) = \mu_c^\sharp(c, \alpha_\Pi(R))$$

Another case analysis reveals that $\mu_t^\sharp$ and $\mu_c^\sharp$ are monotone in their second argument. By a third case analysis one can prove the following lemma concerning applying helper functions to "closed" environments.

Lemma 5 (Helper functions on closed environments)

$$\forall t, R, w : w \in \mu_t^\wp(t, \rho(R)) \implies \{r \mid w \succ^* r\} \subseteq \rho(R)$$
$$\forall c, R, w : w \in \mu_c^\wp(c, \rho(R)) \implies \{r \mid w \succ^* r\} \subseteq \rho(R)$$

Finally we need a lemma formulating abstraction of an extended environment in terms of the abstraction of its subparts.

Lemma 6 (Abstraction of environment extension)

$$\forall v, w, R : \{r \mid w \succ^* r\} \subseteq \rho(R)$$
$$\implies \alpha_\Pi \circ \rho(\{r[v \mapsto w] \mid r \in \rho(R)\}) \mathrel{\dot\subseteq} \alpha_\Pi \circ \rho(R)[v \mapsto \alpha_@(\{w\})]^\sharp$$

For a given program $\lambda k.\, e$:

$$F_\times : \wp(SExp) \times \wp(Env) \to \wp(SExp) \times \wp(Env)$$
$$F_\times(\langle E,\ R\rangle) = \langle \{e\},\ \{\bullet[k \mapsto [\lambda v_r.\, k_r\ v_r,\ \bullet[k_r \mapsto \texttt{stop}]]]\}\rangle$$
$$\cup_\times \bigcup_{\substack{t_0\ t_1\ c \in E \quad r \in R\\ [\lambda x, k'.\, e',\, r'] \in \mu_t^\wp(t_0,\{r\})\\ w \in \mu_t^\wp(t_1,\{r\})\\ w_c \in \mu_c^\wp(c,\{r\})}}{}_\times \langle \{e'\},\ \{r'[x \mapsto w][k' \mapsto w_c]\}\rangle$$
$$\cup_\times \bigcup_{\substack{c\ t \in E \quad r \in R\\ [\lambda v.\, e',\, r'] \in \mu_c^\wp(c,\{r\})\\ w \in \mu_t^\wp(t,\{r\})}}{}_\times \langle \{e'\},\ \{r'[v \mapsto w]\}\rangle$$

Fig. 5. Independent attributes transition function

where we have used the shorthand notation $R^\sharp[v \mapsto \{\ldots\}]^\sharp = R^\sharp \mathbin{\dot\cup} \dot\emptyset[v \mapsto \{\ldots\}]$. We omit the proof due to lack of space.

We are now in position to calculate the abstract transition function. By construction, the abstract transition function satisfies the following lemma. Again, when read directionally from left to right, the lemma states the strategy for the calculation. The resulting analysis appears in Figure 6.

Lemma 7. $\forall S : \alpha_\otimes(F_\times(S)) \subseteq_\otimes F^\sharp(\alpha_\otimes(S))$

Proof. Let $S = \langle E,\, R\rangle$ be given. Again $\alpha_\otimes$ is a complete join morphism and hence distributes onto the three sets from $F_\times$'s definition. We consider the case of the last of the three sets. First observe

$$\begin{aligned}
& r \in R \wedge [\lambda v.\, e',\, r'] \in \mu_c^\wp(c, \{r\}) \wedge w \in \mu_t^\wp(t, \{r\}) \\
\Longrightarrow\; & r \in \rho(R) \wedge [\lambda v.\, e',\, r'] \in \mu_c^\wp(c, \{r\}) \wedge w \in \mu_t^\wp(t, \{r\}) && (\rho \text{ extensive}) \\
\Longrightarrow\; & [\lambda v.\, e',\, r'] \in \mu_c^\wp(c, \rho(R)) \wedge w \in \mu_t^\wp(t, \rho(R)) && (\mu_c^\wp,\, \mu_t^\wp \text{ monotone}) \\
\Longrightarrow\; & r' \in \rho(R) \wedge [\lambda v.\, e',\, r'] \in \mu_c^\wp(c, \rho(R)) \wedge w \in \mu_t^\wp(t, \rho(R)) && (\text{by Lemma 5}) \\
\Longrightarrow\; & r' \in \rho(R) \wedge \alpha_@(\{[\lambda v.\, e',\, r']\}) \subseteq \alpha_@(\mu_c^\wp(c, \rho(R))) \wedge w \in \mu_t^\wp(t, \rho(R)) \\
& && (\alpha_@ \text{ monotone}) \\
\Longleftrightarrow\; & r' \in \rho(R) \wedge [\lambda v.\, e'] \in \alpha_@(\mu_c^\wp(c, \rho(R))) \wedge w \in \mu_t^\wp(t, \rho(R)) && (\text{def. of } \alpha_@) \\
\Longleftrightarrow\; & r' \in \rho(R) \wedge [\lambda v.\, e'] \in \mu_c^\sharp(c, \alpha_\Pi \circ \rho(R)) \wedge w \in \mu_t^\wp(t, \rho(R)) && (\text{by Lemma 4})
\end{aligned}$$

Secondly observe if $w \in \mu_t^\wp(t, \rho(R))$ then

$$\begin{aligned}
& \alpha_\otimes\Big(\bigcup\nolimits_{\times}^{r' \in \rho(R)} \langle \{e'\},\, \{r'[v \mapsto w]\}\rangle\Big) \\
&= \alpha_\otimes(\langle \{e'\},\, \{r'[v \mapsto w] \mid r' \in \rho(R)\}\rangle) && (\text{def. } \cup_\times) \\
&= \langle \{e'\},\, \alpha_\Pi \circ \rho(\{r'[v \mapsto w] \mid r' \in \rho(R)\})\rangle && (\text{def. } \alpha_\otimes) \\
&\subseteq_\otimes \langle \{e'\},\, \alpha_\Pi \circ \rho(R)[v \mapsto \alpha_@(\{w\})]^\sharp\rangle && (\text{by Lemma 5, 6})
\end{aligned}$$

Thirdly observe

$$\begin{aligned}
& \bigcup\nolimits_{\otimes}^{w \in \mu_t^\wp(t, \rho(R))} \langle \{e'\},\, \alpha_\Pi \circ \rho(R)[v \mapsto \alpha_@(\{w\})]^\sharp\rangle \\
&= \langle \{e'\},\, \dot{\bigcup_{w \in \mu_t^\wp(t, \rho(R))}} \alpha_\Pi \circ \rho(R)[v \mapsto \alpha_@(\{w\})]^\sharp\rangle && (\text{def. } \cup_\otimes) \\
&= \langle \{e'\},\, \alpha_\Pi \circ \rho(R) \mathbin{\dot\cup} \dot{\bigcup_{w \in \mu_t^\wp(t, \rho(R))}} \dot\emptyset[v \mapsto \alpha_@(\{w\})]\rangle && (\text{def. } -[-]^\sharp) \\
&= \langle \{e'\},\, \alpha_\Pi \circ \rho(R)[v \mapsto \bigcup_{w \in \mu_t^\wp(t, \rho(R))} \alpha_@(\{w\})]^\sharp\rangle && (\text{def. } \dot\cup) \\
&= \langle \{e'\},\, \alpha_\Pi \circ \rho(R)[v \mapsto \alpha_@(\mu_t^\wp(t, \rho(R)))]^\sharp\rangle && (\alpha_@ \text{ a CJM}) \\
&= \langle \{e'\},\, \alpha_\Pi \circ \rho(R)[v \mapsto \mu_t^\sharp(t, \alpha_\Pi \circ \rho(R))]^\sharp\rangle && (\text{by Lemma 4})
\end{aligned}$$

Hence

$$
\begin{aligned}
&\alpha_{\otimes}\Big(\bigcup_{\times \, \substack{c\,t \in E \quad r \in R \\ [\lambda v.\, e', r'] \in \mu_c^{\wp}(c,\{r\}) \\ w \in \mu_t^{\wp}(t,\{r\})}} \langle \{e'\}, \{r'[v \mapsto w]\}\rangle \Big) \\
&\subseteq_{\otimes} \alpha_{\otimes}\Big(\bigcup_{\times \, \substack{c\,t \in E \quad r' \in \rho(R) \\ [\lambda v.\, e'] \in \mu_c^{\sharp}(c,\alpha_{\Pi}\circ\rho(R)) \\ w \in \mu_t^{\wp}(t,\rho(R))}} \langle \{e'\}, \{r'[v \mapsto w]\}\rangle \Big) && \text{(first obs.)} \\
&= \bigcup_{\otimes \, \substack{c\,t \in E \\ [\lambda v.\, e'] \in \mu_c^{\sharp}(c,\alpha_{\Pi}\circ\rho(R)) \\ w \in \mu_t^{\wp}(t,\rho(R))}} \alpha_{\otimes}\Big(\bigcup_{\times \, r' \in \rho(R)} \langle \{e'\}, \{r'[v \mapsto w]\}\rangle \Big) && (\alpha_{\otimes} \text{ a CJM}) \\
&\subseteq_{\otimes} \bigcup_{\otimes \, \substack{c\,t \in E \\ [\lambda v.\, e'] \in \mu_c^{\sharp}(c,\alpha_{\Pi}\circ\rho(R)) \\ w \in \mu_t^{\wp}(t,\rho(R))}} \langle \{e'\}, \alpha_{\Pi}\circ\rho(R)[v \mapsto \alpha_{@}(\{w\})]^{\sharp}\rangle && \text{(second obs.)} \\
&= \bigcup_{\otimes \, \substack{c\,t \in E \\ [\lambda v.\, e'] \in \mu_c^{\sharp}(c,\alpha_{\Pi}\circ\rho(R))}} \langle \{e'\}, \alpha_{\Pi}\circ\rho(R)[v \mapsto \mu_t^{\sharp}(t, \alpha_{\Pi}\circ\rho(R))]^{\sharp}\rangle && \text{(third obs.)}
\end{aligned}
$$

Since $\alpha_{\otimes}(S) = \langle E, \alpha_{\Pi} \circ \rho(R)\rangle$ we define the third set of $F^{\sharp}$ as this set (with $R^{\sharp}$ for $\alpha_{\Pi} \circ \rho(R)$). By construction $\alpha_{\otimes}(F_{\times}(S)) \subseteq_{\otimes} F^{\sharp}(\alpha_{\otimes}(S))$ holds. □

This result in turn enables us to prove the following (standard) theorem stating the correctness of the analysis [8].

Theorem 1 (Fixed-point transfer theorem). $\alpha_{\otimes} \circ \alpha_{\times}(\text{lfp}\, F_c) \subseteq \text{lfp}\, F^{\sharp}$

The resulting analysis in Figure 6 is striking. By an independent attribute abstraction of a standard collecting semantics, we have encountered a demand-driven CFA. Demand-driven CFA has been discovered independently [2,3,15], and is usually presented as an extension (or improvement) to 0-CFA. Our result on the other hand explains it as a natural abstraction of a reachable states collection semantics.

Expressing a semantics for a CPS language as a transition system lends itself to abstract interpretation as originally expressed by Cousot [5]. Since all intermediate results in CPS are already named, i.e., bound to an identifier, a control-flow analysis merely becomes a question of computing an abstract environment. As both the continuations and closures live in the environment there is no need for an explicit stack, as it lives as a chain of closures in the environment. We have found no need to introduce new concepts such as *labels* or *caches* [4,27].

8 Implementation and Example

We have implemented a prototype of the derived analysis in OCaml.[4] The core of the algorithm constitutes 60 lines of source code. To illustrate the 0-CFA

[4] Available at `http://www.brics.dk/~jmi/Midtgaard-Jensen:SAS08/`

$$\begin{aligned}
\mu_t^\sharp &: \mathit{TExp} \times \mathit{Env}^\sharp \to \wp(\mathit{Val}^\sharp)\\
\mu_t^\sharp(x, R^\sharp) &= R^\sharp(x)\\
\mu_t^\sharp(v, R^\sharp) &= R^\sharp(v)\\
\mu_t^\sharp(\lambda x, k.\, e, R^\sharp) &= \{[\lambda x, k.\, e]\}
\end{aligned}
\qquad
\begin{aligned}
\mu_c^\sharp &: \mathit{CExp} \times \mathit{Env}^\sharp \to \wp(\mathit{Val}^\sharp)\\
\mu_c^\sharp(k, R^\sharp) &= R^\sharp(k)\\
\mu_c^\sharp(\lambda v.\, e, R^\sharp) &= \{[\lambda v.\, e]\}
\end{aligned}$$

(a) Abstract helper functions

For a given program $\lambda k.\, e$:

$$F^\sharp : \wp(\mathit{SExp}) \times \mathit{Env}^\sharp \to \wp(\mathit{SExp}) \times \mathit{Env}^\sharp$$

$$\begin{aligned}
F^\sharp(\langle E^\sharp,\, R^\sharp\rangle) = {} & \langle \{e\},\, \dot{\emptyset}[k_r \mapsto \{\mathtt{stop}\}, k \mapsto \{[\lambda v_r.\, k_r\, v_r]\}]^\sharp\rangle\\
& \cup_\otimes \bigcup_{\substack{t_0\ t_1\ c \,\in\, E^\sharp\\ [\lambda x, k'.\, e'] \,\in\, \mu_t^\sharp(t_0, R^\sharp)}}{}_{\otimes} \langle \{e'\},\, R^\sharp[x \mapsto \mu_t^\sharp(t_1, R^\sharp), k' \mapsto \mu_c^\sharp(c, R^\sharp)]^\sharp\rangle\\
& \cup_\otimes \bigcup_{\substack{c\ t \,\in\, E^\sharp\\ [\lambda v.\, e'] \,\in\, \mu_c^\sharp(c, R^\sharp)}}{}_{\otimes} \langle \{e'\},\, R^\sharp[v \mapsto \mu_t^\sharp(t, R^\sharp)]^\sharp\rangle
\end{aligned}$$

(b) Abstract transition function

Fig. 6. Demand-driven 0-CFA

$k_r \mapsto \{\mathtt{stop}\}$	$f \mapsto \{[\lambda x, k_6.\, k_6\, x]\}$
$k_0, k_2, k_5 \mapsto \{[\lambda v_r.\, k_r\, v_r]\}$	$y \mapsto \{[\lambda y, k_5.\, k_5\, y]\}$
$k_6 \mapsto \{[\lambda v_4.\, v_4\, (\lambda y, k_5.\, k_5\, y)\, k_2], [\lambda v_r.\, k_r\, v_r]\}$	$x, v_4, v_r \mapsto \{[\lambda x, k_6.\, k_6\, x], [\lambda y, k_5.\, k_5\, y]\}$

Fig. 7. Inferred abstract environment

nature of the analysis we recall an example from Nielson, Nielson, and Hankin's textbook [26]: `let f = (fn x => x) in f f (fn y => y)`. The expression $\lambda k_0.\, (\lambda f, k_2.\, f f\, (\lambda v_4.\, v_4\, (\lambda y, k_5.\, k_5\, y)\, k_2))\, (\lambda x, k_6.\, k_6\, x)\, k_0$ is a CPS version of the same example. After 8 iterations the analysis reaches a fixed point determining that all serious expressions of the example are reachable. The inferred abstract environment is given in Figure 7. Just as the textbook 0-CFA the derived analysis merges all bindings to x, which affects the final answer v_r, and results in the overly approximate answer of two abstract closures.

9 Related Work

The only existing demand-driven 0-CFA for a CPS language that we are aware of is that of Ayers [2], who used Galois connections to express the correctness of 0-CFA for a CPS language. After formally proving his 0-CFA correct he suggests a number of improvements. One of these is *use-maps*, the idea of which is to only re-analyse parts of the program where recent additions will have an effect on the fixed-point computation. A later refinement of use-maps incorporates reachability, resulting in an algorithm which will only re-analyse *reachable* parts of the program where recent additions will have an effect. Ayers's work differs from our result in that: (a) it does not use an off-the-shelf starting point,[5] (b) it does not use off-the-shelf Galois connections, and (c) reachability is added afterwards as an extension (but not formally proved, e.g., expressed with Galois connections).

Cousot and Cousot have championed the calculational approach to program analysis for three decades [8,9,10,11,6]. Cousot has provided a comprehensive set of lecture notes [6], in which he calculates various abstract interpreters for a simple imperative language. Nevertheless, the calculational approach to control-flow analysis of functional programs has received little attention so far.

Shivers [31,32,22] has long argued that basing a CFA on a CPS language simplifies matters as it captures all control flow in one unifying construct. In his thesis [32] he developed control-flow analyses for Scheme including (control and state) side effects. Shivers [32] did not consider demand-driven analysis, nor formulate correctness using Galois connections. Initially the development was based on an instrumented denotational semantics, however the more recent work with Might is based on instrumented abstract machines [22]. In contrast we have developed an analysis starting from a well-known and non-instrumented abstract machine.

Sabry and Felleisen [28] have formulated interpreters and corresponding program analysers for languages in direct style and CPS to compare formally their output when run on equivalent input. Their analyses are formulated as inference rules, and as such an analysis may diverge when implemented directly. They therefore detect loops in the analyser and return top when encountering one. In contrast our calculated analysis needs no such ad-hoc modifications. For a further discussion of related work we refer to a recent survey by the first author [21].

10 Conclusion and Further Work

To the best of our knowledge we have given the first calculated 0-CFA derivation. The calculations reveal a strikingly simple derivation of a demand-driven 0-CFA, a variant which has been discovered independently. Our derivation spells out the approximation by expressing it as a combination of several known Galois connections, thereby capturing the essence of the CFA approximation as an independent attributes abstraction. We have derived the analysis from the reachable states of a well-known abstract machine without resorting to instrumentation.

Cousot and Cousot [11] have pointed out several alternative abstractions to sets of pairs, one of which is a *pointwise coding*. When applied to the states of the CE machine the abstraction may be the key to calculating a *flow-sensitive* CFA. We plan to investigate such a calculation. A natural next step is to consider the calculation of the *context-sensitive* k-CFA hierarchy. Finally it would be interesting to investigate whether proof assistants can aid in the calculation of future analyses.

[5] Though to be fair, our starting point, the CE machine in Flanagan et al. [14], and Ayers's thesis are both from 1993.

Acknowledgements. The authors thank Tiphaine Turpin, Mads Sig Ager, Tristan Le Gall, David Pichardie, and the anonymous referees for their comments on this paper.

References

[1] Appel, A.W.: Compiling with Continuations. Cambridge University Press, New York (1992)

[2] Ayers, A.E.: Abstract Analysis and Optimization of Scheme. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts (September 1993)

[3] Biswas, S.K.: A demand-driven set-based analysis. In: Jones (ed.) [17], pp. 372–385.

[4] Bondorf, A.: Automatic autoprojection of higher-order recursive equations. Science of Computer Programming 17(1-3), 3–34 (1991)

[5] Cousot, P.: Semantic foundations of program analysis. In: Muchnick, Jones (eds.) [24], ch. 10, pp. 303–342.

[6] Cousot, P.: The calculational design of a generic abstract interpreter. In: Broy, M., Steinbrüggen, R. (eds.) Calculational System Design. NATO ASI Series F. IOS Press, Amsterdam (1999)

[7] Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. Theoretical Computer Science 277(1–2), 47–103 (2002)

[8] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Sethi, R. (ed.) Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages, Los Angeles, California, pp. 238–252 (January 1977)

[9] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Rosen, B.K. (ed.) Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, pp. 269–282 (January 1979)

[10] Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation 2(4), 511–547 (1992)

[11] Cousot, P., Cousot, R.: Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In: Bal, H. (ed.) Proceedings of the Fifth IEEE International Conference on Computer Languages, Toulouse, France, pp. 95–112 (May 1994)

[12] Danvy, O., Dzafic, B., Pfenning, F.: On proving syntactic properties of CPS programs. In: Third International Workshop on Higher-Order Operational Techniques in Semantics, Paris, France. Electronic Notes in Theoretical Computer Science, vol. 26, pp. 19–31 (September 1999)

[13] Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order, 2nd edn. Cambridge University Press, Cambridge (2002)

[14] Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Wall, D.W. (ed.) Proceedings of the ACM SIGPLAN 1993 Conference on Programming Languages Design and Implementation, Albuquerque, New Mexico, pp. 237–247 (June 1993)

[15] Gasser, K.L.S., Nielson, F., Nielson, H.R.: Systematic realisation of control flow analyses for CML. In: Tofte, M. (ed.) Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming, Amsterdam, The Netherlands, pp. 38–51 (June 1997)

[16] Jones, N.D.: Flow analysis of lambda expressions (preliminary version). In: Proceedings of the 8th Colloquium on Automata, Languages and Programming, London, UK, pp. 114–128 (1981)

[17] Jones, N.D. (ed.): Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Programming Languages, Paris, France (January 1997)
[18] Jones, N.D., Muchnick, S.S.: Complexity of flow analysis, inductive assertion synthesis and a language due to Dijkstra. In: Muchnick, Jones (eds.) [24], pp. 380–393.
[19] Jones, N.D., Nielson, F.: Abstract interpretation: a semantics-based tool for program analysis. In: Handbook of logic in computer science, vol. 4, pp. 527–636. Oxford University Press, Oxford (1995)
[20] Landin, P.J.: The mechanical evaluation of expressions. The Computer Journal 6(4), 308–320 (1964)
[21] Midtgaard, J.: Control-flow analysis of functional programs. Technical Report BRICS RS-07-18, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark (December 2007)
[22] Might, M., Shivers, O.: Improving flow analyses via ΓCFA: abstract garbage collection and counting. In: Lawall, J. (ed.) Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP 2006), Portland, Oregon, pp. 13–25 (September 2006)
[23] Milner, R., Tofte, M.: Co-induction in relational semantics. Theoretical Computer Science 87(1), 209–220 (1991)
[24] Muchnick, S.S., Jones, N.D. (eds.): Program Flow Analysis: Theory and Applications. Prentice-Hall, Englewood Cliffs (1981)
[25] Nielson, F., Nielson, H.R.: Infinitary control flow analysis: a collecting semantics for closure analysis. In: Jones (ed.) [17], pp. 332–345
[26] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)
[27] Palsberg, J.: Closure analysis in constraint form. ACM Transactions on Programming Languages and Systems 17(1), 47–62 (1995)
[28] Sabry, A., Felleisen, M.: Is continuation-passing useful for data flow analysis? In: Sarkar, V. (ed.) Proceedings of the ACM SIGPLAN 1994 Conference on Programming Languages Design and Implementation, Orlando, Florida, pp. 1–12 (June 1994)
[29] Schmidt, D.A.: Denotational Semantics: A Methodology for Language Development. Allyn and Bacon, Inc. (1986)
[30] Sestoft, P.: Replacing function parameters by global variables. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark (October 1988)
[31] Shivers, O.: Control-flow analysis in Scheme. In: Schwartz, M.D. (ed.) Proceedings of the ACM SIGPLAN 1988 Conference on Programming Languages Design and Implementation, Atlanta, Georgia, pp. 164–174 (June 1988)
[32] Shivers, O.: Control-Flow Analysis of Higher-Order Languages or Taming Lambda. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Report CMU-CS-91-145 (May 1991)

Heap Decomposition for Concurrent Shape Analysis

R. Manevich[1,⋆], T. Lev-Ami[1,⋆⋆], M. Sagiv[1], G. Ramalingam[2], and J. Berdine[3]

[1] Tel Aviv University
{rumster,msagiv,tla}@post.tau.ac.il
[2] Microsoft Research India
grama@microsoft.com
[3] Microsoft Research Cambridge
jjb@microsoft.com

Abstract. We demonstrate shape analyses that can achieve a state space reduction exponential in the number of threads compared to the state-of-the-art analyses, while retaining sufficient precision to verify sophisticated properties such as linearizability. The key idea is to abstract the global heap by decomposing it into (not necessarily disjoint) subheaps, abstracting away some correlations between them. These new shape analyses are instances of an analysis framework based on heap decomposition. This framework allows rapid prototyping of complex static analyses by providing efficient abstract transformers given user-specified decomposition schemes. Initial experiments confirm the value of heap decomposition in scaling concurrent shape analyses.

1 Introduction

The problem of verifying concurrent programs that manipulate heap-allocated data structures is challenging: it requires considering arbitrarily interleaved threads manipulating unbounded data structures. Both heap-allocated data structures and concurrency can introduce state explosion. Their combination only makes matters worse. This paper develops new static analysis algorithms that address the state space explosion problem in a systematic and generic way. The result of these analyses can be used to automatically establish interesting properties of concurrent heap-manipulating programs such as the absence of null dereferences, the absence of memory leaks, the preservation of data structure invariants, and *linearizability* [7].

The Intuition. Typical programs manipulate a large number of (instances of) data structures (possibly nested within other data structures). Each individual data structure can usually be in one of several different states (even in an abstract representation). This can lead to a combinatorial explosion in the number of distinct abstract states that can arise during abstract interpretation.

The essential idea we pursue is that of *decomposing* the heap into multiple subheaps and abstracting away some correlations between the subheaps. Decomposition allows reusing subheaps that were decomposed from different heaps, thus representing a set of

⋆ This research was partially supported by the Clore Fellowship Programme.
⋆⋆ Supported by an Adams Fellowship through the Israel Academy of Sciences and Humanities.

© Springer-Verlag Berlin Heidelberg 2008

heaps more compactly (and more abstractly). For example, consider a program maintaining k disjoint lists. A powerset-based shape analysis such as the one in [14] uses a lattice whose height is exponential in k. An abstraction that ignores the correlations between the k lists reduces the lattice height to be linear in k, leading to exponentially faster analysis. (The savings come from not maintaining the correlations between different states of the different lists, which we observe are often irrelevant for a specific property of interest.) Similar situations arise in the kind of multithreaded programs discussed earlier, where the size of the state space is a function of the number of threads rather than the number of data structures. In this paper, we allow decomposing the heap into non-disjoint (i.e., overlapping) subheaps, which is important for handling programs with fine-grained concurrency (where different threads can simultaneously access the same objects) in a thread-modular way.

Fine-Grained Concurrency. Fine-grained concurrent heap-manipulating programs allow multiple threads to use the same data structure *simultaneously*. They trade the simplicity of the single-thread-owning-a-data-structure model, which is at the heart of the coarse-grained concurrency approach, to achieve a higher degree of concurrency. However, the additional performance comes with a price: these programs are notoriously hard to develop and prove correct, even when the manipulated data structures are singly-linked lists (see, e.g., [3]).

It is hard to employ thread-modular approaches that exploit locking [5] to analyze fine-grained concurrent programs because they have *intentional* (benign) data-races. Thus, state-of-the-art shape analyses capable of verifying intricate properties of fine-grained concurrent heap-manipulating programs, e.g., linearizability (explained in Sec. 3), track all correlations between the states of all the threads [1]. This makes these analyses hard to scale. For example, the shape analysis in [1] handles at most 3 threads.

It is interesting to observe, however, that it is often the case that although proving properties of these programs requires tracking sophisticated correlations between every thread and the part of the heap that it manipulates, the correlations between the states of different threads is often irrelevant. Intuitively, this is because fine-grained concurrent programs are often written in a way which *attempts* to ensure the correct operation of every thread *regardless* of the actions taken by other threads. This programming paradigm makes these programs an ideal match with our approach explained below.

The Conceptual Framework. To permit the use of heap decomposition in several settings, we first present it as a parametric abstraction that can be tuned by the analysis designer in three ways:

Decomposition: Specify along what lines a concrete heap should be decomposed into (possibly overlapping) subheaps. One of the strengths of the specification mechanism is that the decomposition of a heap depends on its properties. This allows us, for example, to decompose the state of a concurrent program based on the association between threads and data-structures in that state, which is usually not known a priori.

Subheap abstraction: Create a bounded abstract heap representation from concrete subheaps (which are unbounded). Subheap abstractions can be obtained from existing whole-heap abstractions that satisfy certain properties.

Combiner Sets: The framework is parametric with respect to transformers. Computing sound and precise transformers for statements is quite challenging with a heap decomposition. Transforming each subheap independently can end up being very imprecise (or potentially incorrect, if not done carefully), especially when subheaps overlap. At the other extreme, combining subheaps together into a full heap prior to transforming it can be very inefficient and defeats the purpose of using heap decomposition. Achieving the desired precision and efficiency, without compromising soundness, can be tricky. Our framework allows the analysis designer to specify only which subheaps should be combined together for a given transformer, called combiner sets. The framework automatically generates a corresponding sound transformer, letting the analysis designer easily explore alternatives without worrying about soundness.

HeDec. We implemented our conceptual framework for the family of canonical abstractions [14] in a system called HeDec (for **He**ap **Dec**omposition), which is publicly available. This implementation retains the parametricity of the conceptual framework, which allows analysis designers to rapidly prototype different shape analysis algorithms by defining heap decomposition schemes.

Instances of the Framework. We have used our framework to develop several shape analyses, including the following, and have implemented these analyses in HeDec.

(a) A shape analysis for sequential programs manipulating singly-linked lists that abstracts away the correlations between disjoint lists . The resultant shape analysis algorithm emulates the algorithm of [9], with some interpretative overhead. Unlike the tedious proof of soundness of [9], the soundness of this instance immediately follows from the soundness of the underlying subheap abstraction.

(b) A new shape analysis for sequential programs manipulating singly-linked lists and trees by abstracting away the correlations between segments which do not contain an element pointed-to by a variable. We confirmed that it is precise enough to prove memory safety and preservation of data-structure invariants. This is encouraging for scaling shape analysis for programs with densely connected heaps.

(c) A shape analysis for fine-grained concurrent programs with a bounded number of threads which is precise enough to prove memory safety and preservation of data-structure invariants. Here, we obtain exponential speed-up in terms of time and space, in comparison to similar whole-heap analysis without decomposition. Our algorithm goes beyond [5] by supporting fine-grained concurrency and handling programs with intentional data races.

(d) A shape analysis algorithm for concurrent programs with a bounded number of threads that manipulate singly-linked lists, which proves linearizability. The resultant algorithm is exponentially faster than the one in [1], being polynomial in the number of threads. Our initial empirical results confirm that our algorithm is able to prove linearizability with 20 threads, ten times more than in [1].

Main Results. The contributions of this paper can be summarized as follows:

1. We present a generic analysis framework (in an abstract interpretation setting) for exploiting state decomposition effectively. The main technical contributions are in introducing a family of sound abstract transformers that admit flexibly exploring the efficiency/precision spectrum.
2. We propose scalable analyses for several interesting problems involving coarse-grained as well as fine-grained concurrency, including proving linearizability. These algorithms scale much better (e.g., polynomially) over the number of threads than the previous algorithms for these problems.

3. The implementation of the framework for canonical abstraction is publicly available, together with the above mentioned analyses, as well as other benchmarks, which show the benefit of the approach.

Outline of the Paper. In Sec. 2, we demonstrate heap decomposition for fine-grained concurrent programs. In Sec. 3, we describe an analysis based on heap decomposition for proving linearizability of non-blocking data structures. In Sec. 4 we present the technical details of our abstract domain and its transformers. In Sec. 5 we report on our experiments with HeDec. In Sec. 6, we discuss related work, and in Sec. 7, we conclude the paper.

An accompanying technical report [10] contains proofs and further details.

2 Heap Decomposition for Fine-Grained Concurrency

In this section, we develop decomposition schemes for performing shape analysis of fine-grained concurrent programs and show that HeDec can be used to automatically obtain shape analysis implementations that are precise enough to prove the desired properties of programs (the absence of null pointer dereferences, absence of memory leaks, and data structure invariants) while scaling up to a large number of threads. The material in this section is presented informally, deferring formal definitions and technical details to Sec. 4.

2.1 Decomposing Non-blocking Implementations

A Running Example. Fig. 1 shows a simple running example of a non-blocking stack implementation from [15]. Producers push elements onto the stack by allocating an element, copying the current global pointer to the top of the stack, connecting the new element to that copied top, and then using CAS (**C**ompare **A**nd **S**wap) to atomically check that the top of the stack has not changed and replace it with the new element. Consumers pop elements from the stack by copying the current global pointer to top and recording its next element and then using CAS to atomically check that the top

```
#define EMPTY -1
typedef int data_type;
typedef struct node t {
      data_type d;
      struct node t *n
} Node;
typedef struct stack t {
      struct node t *Top;
} Stack;

void push(Stack *S, data_type v){
  Node *x = alloc(sizeof(Node));
  x->d = v;
  do {
    Node *t = S->Top;
    x->n = t;
  } while (!CAS(&S->Top,t,x));
}
```

```
data_type pop(Stack *S){
  do {
    Node *t = S->Top;
    if (t == NULL)
      return EMPTY;
    Node *s = t->n;
    data_type r = t->d;
  } while (!CAS(&S->Top,t,s));
  return r;
}
```

Fig. 1. A non-blocking stack implementation

of the stack has not changed and replace it with the new top, i.e., the recorded next element. In both cases, a failed CAS results in a restart.

The goal here is to prove the absence of null pointer dereferences, absence of memory leaks, and the preservation of data structure invariants, i.e., that `stack` points to an acyclic list.

Concrete Execution. Fig. 2(a) shows an example of two states occurring in the non-blocking implementation shown in Fig. 1; for now ignore the $corr$ annotations (which is used by the linearizability analysis in the next section). The figure shows two consumer threads and two producer threads. Both **cons1** and **prod1** can succeed with the CAS if they are the next threads to be scheduled. Concrete states are depicted by graphs. To avoid clutter the `data` field is not shown. Hexagonal nodes denote thread objects and square nodes denote list elements. The program label of every thread is written inside the hexagon. Edges from text labels to nodes correspond to global pointers (`Top`). Labeled edges from thread nodes to list nodes denote thread-local pointer variables (`t` and `x`). Edges between list nodes, labeled by `n` correspond to the `next` field of the list.

Exponential State Space. There are several sources of exponential explosion in the state space exploration of the stack algorithm. The first one is the correlation between the program locations of the different threads. The second source is the next pointers of the just allocated elements. The stack can grow after the next pointer has already been set, but before the CAS, thus the next pointers of the different producers can point to all possible stack elements and have all possible aliasing between each other. The third source of state-space explosion is the recorded next pointer of the consumer threads. Note that the state space explosion occurs even if the list has a bounded number of elements. This is a general problem when maintaining correlations between the properties of different threads. Exponential blow-ups also occur in sequential programs because of aliasing. However, for the purpose of our analysis, these correlations are unimportant and tracking them is pointless and only reduces the efficiency of the analysis.

Heap Decomposition Abstraction. We reduce the size of the state space by decomposing the heap into a set (or tuple) of subheaps and abstractly interpreting the program over the subheaps.

For each subheap to be used in the decomposition, a user of HeDec specifies the part of the heap it should include. This is done by defining a *location selection predicate*, which specifies the subset of the nodes in the state for which abstract properties (such as aliasing, heap-reachability, etc.) are maintained. For each location selection predicate, the program state is projected onto the nodes satisfying that predicate, thus obtaining a *substate* of the original state. We refer to the domain of substates pertaining to a location selection predicate pt as the *subdomain* of pt.

The Decomposition Scheme. For the purpose of our analysis, we define for each thread t the location selection predicate $pt[t]$ that holds for: (a) the thread object of t, (b) the objects pointed-to by its local variables (`t` and `x`), and (c) the objects pointed-to by the global variables (`Top`). In addition, we define the location selection predicate *Globals*, which holds for the objects reachable from global variables.

Fig. 2(b) shows the result of applying the decomposition scheme explained above to the states in Fig. 2(a). Notice that different location selection predicates may

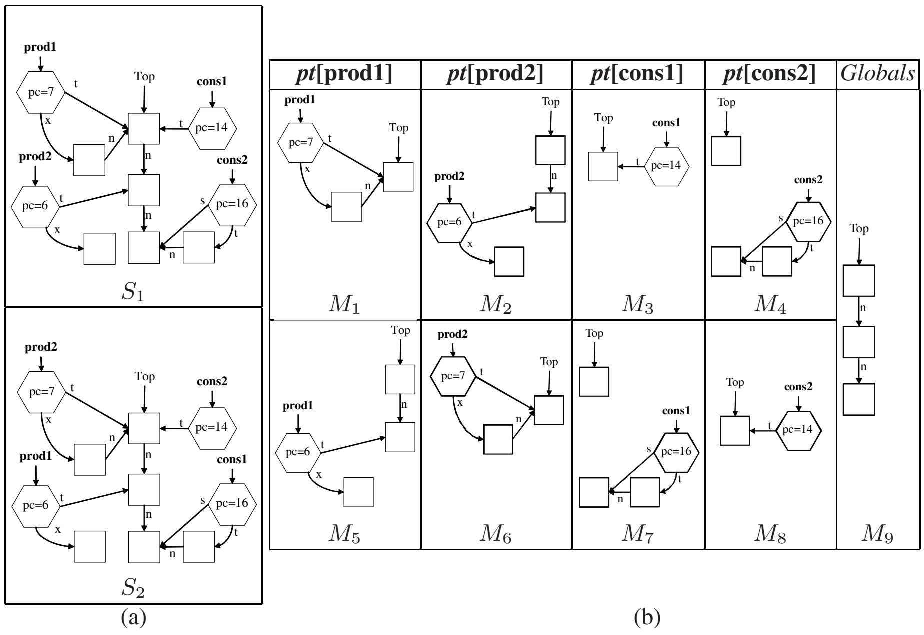


Fig. 2. (a) Two concrete states in the non-blocking stack implementation shown in Fig. 1; and (b) The decomposed states abstracting the full states in (a). The names of the sub-domains appear above the substates.

occasionally overlap. For example, in the decomposition explained above, the objects reachable from the global variables appear in each subheap.

Intuitively, the meaning of a substate M, decomposed by a location selection predicate $p(v)$, is the set of all full states that contain M and any disjoint substate M', such that the objects in M satisfy $p(v)$ and the objects in M' do not satisfy $p(v)$. A sequence of sets of substates $\{M_1, M_5\} \times \{M_2, M_6\} \times \{M_3, M_7\} \times \{M_4, M_8\} \times \{M_9\}$ represents the set of full states obtained by choosing one structure from each subdomain and intersecting their meanings. For example, composing the substates $\{M_1, M_2, M_3, M_4, M_9\}$ together yields S_1 and composing the substates $\{M_5, M_6, M_7, M_8, M_9\}$ together yields S_2. The loss of precision by the abstraction can be observed by the fact that other compositions, such as $\{M_1, M_6, M_7, M_8, M_9\}$ yield full states other than S_1 and S_2.

State Space Savings. In general, for n threads, if the set of objects reachable from a thread is bounded, then the number of substates resulting from the reachability-based decomposition is linear in n (even though the number of full states generated by the program is exponential in n). Although we do not show the state space reduction in the figures, one can imagine how running the program with n threads generates states similar to the ones in Fig. 2(a). By permuting the thread ids between producers threads and between consumer threads, we obtain an exponential number of full states that are all reachable by the program execution. Decomposing these states results in a number of substates that is linear in n.

Transformers. HeDec is guaranteed to be sound, in the sense that when the analysis terminates all reachable concrete states are represented by some abstract state.

While the abstraction ignores correlations between substates, transforming substates in isolation using an "independent-attribute" style of analysis [13] leads to debilitating loss of precision. For example, the analysis executes the statement `6: x->n=t` where thread **prod1** is scheduled. Substate M_3 does not contain information about the local variables of thread **prod1**. Therefore, M_3 also represents a state S_{bad} in which the local variables `t` and `x` of thread **prod1** point to the first cell and to the last cell of the list, respectively. Thus, a conservative transformer of `6: x->n=t` must emit a warning about a possible creation of a cyclic list.

To avoid this kind of loss of precision, a user of HeDec can specify which substates, obtained from different location selection predicates, should be (temporarily) composed by the transformer. This is done in terms of *combiner sets*, which are subsets of node selection predicates. In this example, for the transformer of `6: x->n=t`, we can specify the combiner sets $\{pt[\textbf{prod1}], pt[\textbf{prod2}]\}$, $\{pt[\textbf{prod1}], pt[\textbf{cons1}]\}$, $\{pt[\textbf{prod1}], pt[\textbf{cons2}]\}$, and $\{pt[\textbf{prod1}], pt[Globals]\}$. Then, the generated transformer composes, separately, the substates $\{M_1, M_5\}$ with each of the sets of substates $\{M_2, M_6\}$, $\{M_3, M_7\}$, $\{M_4, M_8\}$, and $\{M_9\}$. For the substates composed with M_5 (which is the only substate in the **prod1**-subdomain that can execute `6: x->n=t`) the transformer updates the `n` field appropriately, avoiding the false alarm. Finally, the transformer decomposes the substates again into each one of the subdomains. The resulting abstract substates are the same as in Fig. 2, except that M_5 has an `n`-link between the object pointed-to by `t` and the object pointed-to by `x` and its program counter is 7.

This example shows how, by combining a small number (linear in the number of location selection predicates, in this case) of substates decomposed by different predicates, the transformer is able to increase precision without incurring an unreasonable time/space blow-up.

A Methodology for Combiner Sets. We now briefly discuss the issue of choosing combiner sets for a transformer (which is done by the analysis designer in our framework). Every transformer can be thought of as having a *frame* as well as a *footprint*. The frame identifies the part of a program state that is completely irrelevant to the transformer. Thus, it contains no information that is either used or modified by the transformer. The footprint is the complement and contains adequate information to perform the transformer as precisely as possible.

A straightforward approach for computing the footprint of an operation affecting several subdomains is combining all the affected subdomains. Unfortunately, this approach might be too expensive. We apply a more efficient approach, which according to our experience is precise enough. Specifically, for each operation we choose a set of *core subdomains* which contain the heap objects and variables that participate in the operation. We compute the *core footprint* by combining the core subdomains (in practice, there are usually no more than two). We then independently combine the core footprint with the other affected subdomains. For example, the core subdomains for a statement of the form "`x->f = g`", where `x` of thread t is a local variable and `g` is a global variable, are the subdomains containing thread t and the subdomain of the global variable `g`. The affected subdomains are any subdomains which may alias these variables.

Conditional branches pose an interesting puzzle. Note that because the condition essentially filters states it can affect *all* subdomains. Thus, for a conditional "`if (x == g)`", we identify the core subdomains to be the ones containing (the nodes pointed-to by) `x` and `g`. However, we will independently combine them with all other subdomains.

3 Using Decomposition to Prove Linearizability

Linearizability [7] is one of the main correctness criteria for implementations of concurrent data structures. Informally, a concurrent data structure is said to be linearizable if the concurrent execution of a set of operations on it is equivalent to some sequential execution of the same operations, in which the global order between non-overlapping operations is preserved. The equivalence is based on comparing the arguments and results of operations (responses). The permitted behavior of the concurrent object is defined in terms of a specification of the desired behavior of the object in a sequential setting. Linearizability is a widely-used concept, and there are numerous non-automatic proofs of linearizability for concurrent objects.

Verifying linearizability is challenging because it requires correlating any concurrent execution with a corresponding permitted sequential execution. Verifying linearizability for concurrent dynamically allocated linked data structures is particularly challenging, because it requires correlating executions that may manipulate memory states of unbounded size. Interestingly, proving linearizability does not require directly proving safety properties such as preservation of data structure invariants. Instead, one can first prove that the sequential implementation satisfies the required safety properties and then prove that the concurrent implementation is linearizable, thereby, satisfies the safety property. Finally, linearizability of complex systems can be shown by separately proving the linearizability of each of the individual data structure implementations.

Intuitively, we verify linearizability by representing, in the concrete state, both the state of the concurrent program and the state of the reference sequential program. Each element entered into the data structure is correlated at linearization points with the matching object from the sequential execution. This works well under abstraction when the differences between the heaps of the sequential and concurrent implementations are bounded. The details are described in [1].

In order to guarantee that the shape analysis scales-up in the number of threads, in HeDec we have defined a decomposition scheme that abstracts away the correlations between the threads (as in Sec. 2). Also, there is no need to track reachability from program variables. Instead, the subheap abstraction tracks elements whose values in the sequential and the concurrent implementations are correlated.

3.1 A Decomposition Scheme for Linearizability Analysis

In HeDec, we have defined such a decomposition scheme by decomposing the heap into $n+1$ components where n is the number of threads: (i) For each thread the objects pointed-to by local variables of the thread and objects pointed-to by global variables. This captures the relationships between local pointer variables and global pointer variables. Each subheap abstracts away the values of the local variables of the other threads. (ii) A separate subheap with the objects pointed-to by global variables and the part of the heap already correlated with the sequential execution. Here, the values of the local

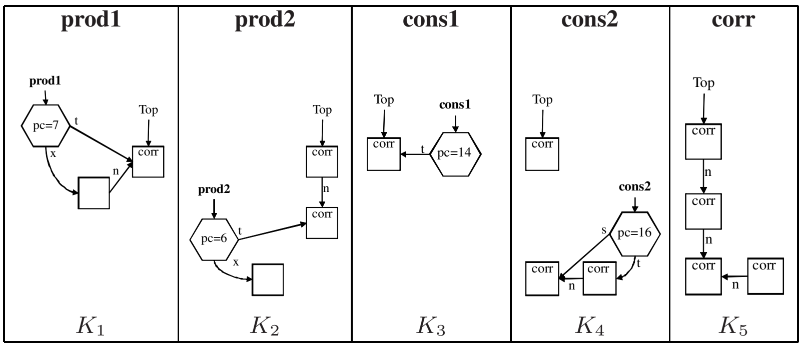


Fig. 3. The decomposed states abstracting the full state S_1 in Fig. 2(a). The names of the subdomains appear above each substate.

variables of all the threads are abstracted away. We call this the *corr* subdomain as it represents the correlated elements. Fig. 3 shows the effect of applying this decomposition to the full state S_1 in Fig. 2(a).

Intuitively, this decomposition is appropriate for verifying linearizability for the program in Fig. 1 because of the following. The list consisting of correlated objects changes locally when a thread executes a successful `CAS` operation. In fact, successful `CAS` operations are the linearization points for this program. Precisely interpreting these operations (`CAS(&S->Top,t,x)` and `CAS(&S->Top,t,s)`) in the analysis requires tracking correlations between local and global variables, which we do in the subheap we decompose for each thread.

The subheap captured by the *corr* subdomain is important only during successful `CAS` operations, which is when a (non-correlated) node allocated by a thread is passed into the list. Maintaining the subheap of the *corr* subdomain for each thread is wasteful, and thus we separate these correlations into different subdomains.

The important thing to notice is that all the exponential explosion in the state space that is due to the number of threads in the full heap is eliminated by this decomposition. The number of possible subheaps of each thread becomes independent of the number of threads in the system (for more than two threads).

Transformers. The combiner sets used in the transformers of the analysis are the application of the methodology described in Sec. 2.1 to this decomposition scheme. For example, copying a global variable into a local variable does not require decomposition as the executing thread has all the needed information. Copying a local variable into a global variable combines the subdomain of the executing thread with each of the other subdomains. Other operations that change the global state such as changes to pointer fields and performing CAS operations behave the same. Dereferencing a pointer requires composing the subdomain for the current thread and the *corr* subdomain as the information on the next element of the stack is not available in the thread's subdomain.

4 The Heap Decomposition Abstraction

In this section, we formally define our new parametric heap abstraction and a family of sound abstract transformers.

4.1 Heap Decomposition as a Cartesian Product of Subheaps

We first define the (parameterized) abstract domain of decomposed heaps. (See the technical report [10] for an illustration of the concepts defined below.)

Let $(\Sigma, \preceq, \otimes)$ be a semilattice, where elements of Σ represent (total and partial) states, $\preceq$ is a partial ordering on Σ capturing the "is a substate of" relation, and $\otimes$ is the join operation with respect to $\preceq$ (which composes substates together). We extend $\otimes$ to sets of states as follows. Let $X_1 \subseteq \Sigma$ and $X_2 \subseteq \Sigma$. We define $X_1 \otimes X_2 = \{\sigma_1 \otimes \sigma_2 \mid \sigma_1 \in X_1, \sigma_2 \in X_2\}$. For purposes of abstraction, we shall also make use of the information ordering defined by $\sigma \sqsubseteq \sigma'$ iff $\sigma' \preceq \sigma$.

Let $(\mathcal{P}(\Sigma), \sqsubseteq)$ denote the powerset domain of Σ with the Hoare ordering: i.e., for every $X, Y \subseteq \Sigma$, we write $X \sqsubseteq Y$ iff $\forall x \in X : \exists y \in Y : x \sqsubseteq y$.

A *substate extraction* function is a function $\eta : \Sigma \to \Sigma$ that satisfies $\eta(\sigma) \preceq \sigma$. Assume we have a sequence of k substate extraction functions η_1 to η_k. We use the k-fold product $\mathcal{P}(\Sigma)^k = \mathcal{P}(\Sigma) \times \cdots \times \mathcal{P}(\Sigma)$ as our domain of abstract states. The abstraction function $\alpha : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)^k$ is defined by:

$$\alpha(S) = (\hat{\eta}_1(S), \ldots, \hat{\eta}_k(S)) \tag{1}$$

where $\hat{\eta}_i$ is the pointwise extension of η_i defined by:

$$\hat{\eta}_i(S) = \{\eta_i(\sigma) \mid \sigma \in S\} \tag{2}$$

We define the meaning, or *concretization*, of a tuple $I_1, \ldots, I_k \in \mathcal{P}(\Sigma)^k$ by

$$\gamma(I_1, \ldots, I_k) = I_1 \otimes \cdots \otimes I_k. \tag{3}$$

Example 1. Let S denote the set of states $\{S_1, S_2\}$ shown in Fig. 2(a). For any thread t, we define the predicate $pt[t]$ to be true for: (a) the thread object of t, (b) the objects pointed-to by its local variables (`t` and `x`), and (c) the objects pointed-to by the global variables (`Top`). In addition, we define the location selection predicate *Globals*, which holds for the objects reachable from global variables. Given any predicate p, the substate extraction function δ_p maps a state σ to the substate consisting only of the locations satisfying p. We define η_1 to be $\delta_{pt[\textbf{prod1}]}$, η_2 to be $\delta_{pt[\textbf{prod2}]}$, η_3 to be $\delta_{pt[\textbf{cons1}]}$, η_4 to be $\delta_{pt[\textbf{cons2}]}$, and η_5 to be $\delta_{Globals}$. Now, $\eta_1(S_1) = M_1$, $\eta_2(S_1) = M_2$, $\eta_3(S_1) = M_3$, $\eta_4(S_1) = M_4$, and $\eta_5(S_1) = M_9$.

4.2 Abstract Transformers

We now turn our attention to the more challenging aspect of decomposition: computing sound abstract transformers.

The semantics of a program statement is given by a function $\tau : \Sigma \to \mathcal{P}(\Sigma)$. We make the standard assumption that the transformer is monotonic in the information order, i.e., if $\sigma_1 \sqsubseteq \sigma_2$ then $\tau(\sigma_1) \sqsubseteq \tau(\sigma_2)$. We extend this function pointwise to $\tau : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$, by defining $\tau(S) = \bigcup\{\tau(\sigma) \mid \sigma \in S\}$. (Note that the extended transformer is monotone in the information order as well.) For purposes of abstract interpretation, we need to define a corresponding sound abstract transformer on $\mathcal{P}(\Sigma)^k$. Given an input value $I = (I_1, \ldots, I_k)$, the abstract transformer needs to compute the output value $O = (O_1, \ldots, O_k)$.

A straightforward sound transformer is the pointwise transformer τ^{pw} defined as follows:

$$\tau^{pw}(I_1, \ldots, I_k) = (\hat{\eta}_1(\tau(I_1)), \ldots, \hat{\eta}_k(\tau(I_k))). \tag{4}$$

Example 2. While the pointwise transformer is simple and efficient, it can lead to imprecise results when the transformer has to update a substate that does not have all the relevant information. Recall the example from Sec. 2, and consider the substate M_3. Substate M_3 does not contain information about the local variables of other threads. Therefore, M_3 also represents a state S_{bad} in which the local variables `t` and `x` of thread **prod1** point to the first cell and to the last cell of the list, respectively. Thus, a conservative transformer of `6: x->n=t`, when **prod1** serves as the scheduled thread, must emit a warning about a possible creation of a cyclic list. As explained in Sec. 2, we can avoid this imprecision by composing substate M_3 with other substates (M_1) to produce a more precise substate that can be transformed without making such worst-case assumptions. This motivates the following definitions.

A *combiner set* is a set $R \subseteq \{1, \ldots, k\}$ identifying a set of subheap domains. We define the *partial concretization function* γ_R, which combines the information from the specified set of subdomains $R = \{j_1, \ldots, j_m\}$, as follows:

$$\gamma_R(I_1, \ldots, I_k) = \bigotimes_{r \in R} I_r = I_{j_1} \otimes I_{j_2} \cdots \otimes I_{j_m} \quad . \tag{5}$$

One-Level Composition. We define the *partial transformer* $\tau_1[R, i]$, which computes the substate corresponding to the i-th subdomain using the subdomains identified by R, by

$$\tau_1[R, i](I) = \hat{\eta}_i(\tau(\gamma_R(I))). \tag{6}$$

We use the term *one-level* transformer to indicate that combining (or composing) information from a set of subdomains (identified by R above) occurs in one step.

We define a *one-level transformer specification* $\textsc{ts}$ to be a tuple $(\textsc{ts}_1, \ldots, \textsc{ts}_k)$ where each $\textsc{ts}_i \subseteq \{1, \ldots, k\}$. We define the transformer $\tau_1[\textsc{ts}]$ by

$$\tau_1[\textsc{ts}](I) = (\tau_1[\textsc{ts}_1, 1](I), \ldots, \tau_1[\textsc{ts}_k, k](I)). \tag{7}$$

Theorem 1. *For any one-level transformer specification* $\textsc{ts}$*, the transformer* $\tau_1[\textsc{ts}]$ *is sound. That is, for every input value* $I \in \mathcal{P}(\Sigma)^k$: $\tau(\gamma(I)) \sqsubseteq \gamma(\tau_1[\textsc{ts}](I))$.

Two-Level Composition. We now present a generalization of the above definition. As motivation for this generalization, consider a situation where we want to compute an output value O_j by combining the input values from a set of subdomains R_1 or by combining the input values from a set of subdomains R_2 (but we are unable to say which of these combinations to use statically). We could, of course, combine the input values from the set of subdomains $R_1 \cup R_2$, but this could be expensive. Instead, we can utilize the two combinations *independently* of each other by using

$$(\hat{\eta}_j(\tau(\gamma_{R_1}(I)))) \sqcap (\hat{\eta}_j(\tau(\gamma_{R_2}(I))))$$

as the desired output value. We call transformers derived in this fashion two-level transformers, as the use of the meet operation $\sqcap$ constitutes a second stage of combining (composing) information.

Let Y be a set of combiner sets. We define the *partial transformer* $\tau_2[Y, i]$, which computes the substate corresponding to the i-th subdomain using the combiner sets in Y independently, as follows:

$$\tau_2[Y, i](I) = \bigsqcap_{R \in Y} \tau_1[R, i](I) \tag{8}$$

We define a *two-level transformer specification* $\textsc{ts}$ to be a tuple $(\textsc{ts}_1, \ldots, \textsc{ts}_k)$ where each $\textsc{ts}_i \subseteq \mathcal{P}(\{1, \ldots, k\})$. We define the transformer $\tau_2[\textsc{ts}]$ by

$$\tau_2[\textsc{ts}](I) = (\tau_2[\textsc{ts}_1, 1](I), \ldots, \tau_2[\textsc{ts}_k, k](I)). \tag{9}$$

(Note that the computation of the above transformer involves a partial concretization for every R in every $\textsc{ts}_i$. In practice, different $\textsc{ts}_i$ and $\textsc{ts}_j$ may have common elements, and it is sufficient for the transformer implementation to do the corresponding partial concretization just once.)

Theorem 2. *For any two-level transformer specification* $\textsc{ts}$*, the transformer* $\tau_2[\textsc{ts}]$ *is sound. That is, for every input value* $I \in \mathcal{P}(\Sigma)^k$*:* $\tau(\gamma(I)) \sqsubseteq \gamma(\tau_2[\textsc{ts}](I))$.

5 Empirical Results

We implemented the HeDec system in Java on top of the TVLA system [8]. HeDec allows analysis designers to rapidly prototype different shape analysis algorithms by defining heap decomposition schemes. HeDec, however, is not a panacea — the designer needs to carefully select suitable heap decompositions. Nevertheless, HeDec relieves the designer from the task of developing and implementing the static analysis algorithms, including the transformers.

Fig. 4 compares the results of our decomposition-based analysis with a full heap analysis.[1]

Concurrent Benchmarks. We use the analysis of [1] as the underlying shape analysis.

Both analyses successfully prove linearizability and absence of null dereferences for the three concurrent programs. For a given number of threads, t, the table shows the time and the number of states resulting in the analysis of t threads invoking an arbitrary sequence of operations on a single instance of the analyzed concurrent data structure. Stack is the non-blocking stack example of Sec. 2.1. TLQ is the two-lock queue implementation described in [12]. NBQ is a non-blocking queue implementation from [4]. [2]

Note that while [1] can analyze at most 3 threads, our approach, on the other hand, runs for 15 threads or more. Furthermore, [1] runs out of memory when analyzing 3 threads manipulating a non-blocking-queue.

[1] All benchmarks except NBQ were run on a 2.4 GHz E6600 Core 2 Duo processor with 2 GB of memory running Linux.

[2] This benchmark was run on a 2.66 GHz Quad Xeon with 16 GB of memory running Windows XP 64 bit.

		Full Heap		Decomposition	
Example	# of threads	# of states	secs.	# of substates	secs.
Stack	2	3,424	3	1,608	7
	3	10,6296	71	4,103	13
	4	MemOut	-	7,728	22
	20	-	-	212,048	3,421
TLQ	3	8,783	12	8,911	30
	5	44,285	35	23,585	90
	8	MemOut	-	58,796	307
	15	-	-	202,555	2,122
NBQ	2	39,583	69	20,646	263
	3	MemOut	-	57,065	694
	15	-	-	2,017,280	1 day

(a)

	Full Heap		Decomposition	
Example	# of states	secs.	# of substates	secs.
6-list-prepend	17,496	16	557	5
6-list-join	37,689	40	1,282	6
4-tree-insert	43,031	44	5,316	29

(b)

Fig. 4. Empirical results for: (a) concurrent benchmarks, and (b) sequential benchmarks

Sequential Benchmarks. Both analyses successfully prove absence of null dereferences, absence of memory leaks, and data structure invariants for the following sequential benchmarks: `6-list-prepend` adds elements, non-deterministically, into one of 6 lists; `6-list-join` joins 6 lists into one list; and `4-tree-insert` inserts nodes, non-deterministically, into one of 4 binary search trees.

6 Related Work

The framework of Cartesian abstraction via state decomposition we have presented is relevant to a number of previous lines of work.

Heterogeneous Abstractions. Yahav and Ramalingam [19] defined a notion of heterogeneous abstractions. There, Cartesian abstractions are used as a way to achieve decomposition (or separation, in the terminology of that paper). One contribution of this paper is to show that that previous analysis is based on a (simple form of) Cartesian abstraction. On the other hand, in that work, heterogeneity was used only within a single structure (to abstract the substructure of interest differently from its context), where our framework supports different abstractions for different factors of the product, yielding heterogeneity across different structures. Furthermore, while [19] relies on the point-wise transformer, we introduce a generalized family of transformers that allow (de)composition when transformers are applied. This generalization allows specifying more precise transformers, and gives us dynamic separation/decomposition.

Region-based Heap Analyses. Like [19], [6] also decomposes heap abstractions to independently analyze different parts of the heap. There the analysis/verification problem is itself decomposed into a set of problem instances, and the heap abstraction is specialized for each instance and consists of one subheap for the part of the heap relevant to the instance, and a coarser abstraction of the remaining part of the heap, e.g. a points-to graph. In contrast, we simultaneously maintain abstractions of different parts of the heap and also consider the interaction between these parts. (E.g., our decomposition dynamically changes as components get connected and disconnected.)

Partially Disjunctive Heap Abstraction. Manevich et al. [11] describe a heap abstraction based on merging sets of graphs with the same set of nodes into one (approximate) graph. The abstraction in this paper is based on decomposing a graph into a set of subgraphs. The abstraction in [11] is orthogonal to the one in this paper.

Handling Concurrency for an Unbounded Number of Threads. In [2], we use thread quantification to analyze programs with an unbounded number of threads. Thread quantification can be thought of as an unbounded variant of a particular decomposition strategy, which we use to abstract away correlations between local variables of different threads. In the thread quantification analysis, we report that using an additional heap decomposition abstraction in order to abstract away correlations between values of some local variables and global variables effects drastic state-space savings. This made the analysis feasible in the example of proving linearizability of a non-blocking queue implementation.

Proving Linearizability of Data Structures. Shape analysis of concurrent programs with unbounded dynamic allocation have been investigated. The analysis in [18] addresses an unbounded number of threads by losing distinctions that cannot be made based on thread-independent information. This analysis has been extended to verify linearization [1] of programs with a bounded number of threads. Here we use the decomposition abstraction to define an analysis that can be exponentially faster than that in [1].

Manual linearizability proofs using rely-guarantee have been given in [17], and using a manual translation to automata followed by an interactive proof in PVS in [2]. Recently, [16] automatically verifies linearizability from manual specifications in a combination of rely-guarantee and separation logic, using the proof technique of [1].

7 Conclusions

We present systematic and generic techniques for scaling up shape analyses using heap decomposition, implemented in the HeDec system. A user of HeDec can quickly prototype a shape analysis by: (a) defining any heap decomposition she believes is appropriate for the class of programs and properties of interest, and (b) supplying for every type of program statement any (possibly empty) combiner set she believes supplies the right balance between efficiency and precision. HeDec then automatically generates a sound analysis.

Acknowledgements. We thank Noam Rinetzky, Greta Yorsh, Byron Cook, and Thomas Ball for supplying us with helpful comments on early drafts of the paper. We thank Daphna Amit for explaining and helping us use her linearizability analysis and commenting on earlier drafts of this paper.

References

1. Amit, D., Rinetzky, N., Reps, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 477–490. Springer, Heidelberg (2007)
2. Colvin, R., Doherty, S., Groves, L.: Verifying concurrent data structures by simulation. Electr. Notes Theor. Comput. Sci. 137(2), 93–110 (2005)

3. Doherty, S., Detlefs, D.L., Groves, L., Flood, C.H., Luchangco, V., Martin, P.A., Moir, M., Shavit, N., Steele Jr., G.L.: DCAS is not a silver bullet for nonblocking algorithm design. In: SPAA, pp. 216–224 (2004)
4. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: Núñez, M., Maamar, Z., Pelayo, F.L., Pousttchi, K., Rubio, F. (eds.) FORTE 2004. LNCS, vol. 3236, pp. 97–114. Springer, Heidelberg (2004)
5. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: PLDI, pp. 266–277 (2007)
6. Hackett, B., Rugina, R.: Region-based shape analysis with tracked locations. In: POPL, pp. 310–323 (2005)
7. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. TOPLAS 12(3), 463–492 (1990)
8. Lev-Ami, T., Sagiv, M.: TVLA: A framework for implementing static analyses. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 280–301. Springer, Heidelberg (2000)
9. Manevich, R., Berdine, J., Cook, B., Ramalingam, G., Sagiv, M.: Shape analysis by graph decomposition. In: TACAS, pp. 3–18 (2007)
10. Manevich, R., Lev-Ami, T., Sagiv, M., Ramalingam, G., Berdine, J.: Heap decomposition for concurrent shape analysis. Technical Report TR-2008-01-85453, Tel Aviv University (January 2008), http://www.cs.tau.ac.il/~rumster/TR-2007-11-85453.pdf
11. Manevich, R., Sagiv, M., Ramalingam, G., Field, J.: Partially disjunctive heap abstraction. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 265–279. Springer, Heidelberg (2004)
12. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC, pp. 267–275 (1996)
13. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999)
14. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems 24(3), 217–298 (2002)
15. Treiber, R.K.: Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center (April 1986)
16. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. draft (2008)
17. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: PPOPP, pp. 129–136 (2006)
18. Yahav, E.: Verifying safety properties of concurrent Java programs using 3-valued logic. ACM SIGPLAN Notices 36(3), 27–40 (2001)
19. Yahav, E., Ramalingam, G.: Verifying safety properties using separation and heterogeneous abstractions. In: PLDI, pp. 25–34 (2004)

Author Index